OAK RIDGE
NATIONAL LABORATORY
MANAGED BY UT-BATTELLE
FOR THE DEPARTMENT OF ENERGY

ORNL/TM-2013/339

Synthetic Graph Generation for Data-Intensive HPC Benchmarking: Background and Framework

October 2013

Prepared by
Joshua Lothian, Sarah Powers, Blair D. Sullivan, Matthew Baker, Jonathan Schrock, Stephen W. Poole

UT-BATTELLE
ORNL-27 (4-00)

ORNL/TM-2013/339

Extreme Scale Systems Center, Computer Science and Mathematics Division

Synthetic Graph Generation for Data-Intensive HPC Benchmarking: Background and Framework

Joshua Lothian, Sarah Powers, Blair D. Sullivan, Matthew Baker, Jonathan Schrock, Stephen W. Poole

Prepared by
OAK RIDGE NATIONAL LABORATORY
P.O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

Abstract

The benchmarking effort within the Extreme Scale Systems Center at Oak Ridge National Laboratory seeks to provide High Performance Computing benchmarks and test suites of interest to the DoD sponsor. The work described in this report is a part of the effort focusing on graph generation. A previously developed benchmark, SystemBurn, allows the emulation of a broad spectrum of application behavior profiles within a single framework. To complement this effort, similar capabilities are desired for graph-centric problems. This report examines existing synthetic graph generator implementations in preparation for further study on the properties of their generated synthetic graphs.

1. INTRODUCTION

A common challenge facing academic, government, and industrial institutions today is that of acquiring appropriate computer hardware to support a growing demand for computational resources in scientific research, national security, and corporate development. In order to maximize return on investment, it is critical for strategic tradeoffs between architectures to be evaluated for factors such as cost, energy consumption, and performance/suitability for the intended applications. In the high performance computing community, benchmarks play an increasingly important role in this process. Specifically designed to mimic access patterns, computational complexity, and communication behavior of target algorithms or applications, benchmarks provide a standardized test which can be used to compare disparate architectures using concrete, objective metrics.

As the extensive literature on benchmarking illustrates (see, for example, [2, 28, 68, 82]), designing (one or more) kernels which reproduce application behaviors closely enough to yield accurate predictions of a machine's future performance is a non-trivial task. In particular, many aspects of execution – and their interactions – must be reflected. For example, a program's performance is affected by the quantity and (relative) timing of its I/O usage, floating point operations, integer operations, fetches to different levels of the memory hierarchy, inter-core and inter-processor communication (for parallel jobs). Traditional benchmarks test performance for each factor independently, (for example, GUPS as tested by RandomAccess in [64] measures the performance of random memory accesses), but this has proven insufficient in the face of increasingly complex workloads and machine heterogeneity. To enable effective evaluation in this environment, recent research [62] has focused on developing tools which allow emulation of a broad spectrum of application behavior profiles within a single framework.

Although these tools play a vital role in guiding development and acquisition of computational infrastructure, they often fall short when target applications are data-intensive, as is often the case when performing sparse network analysis in domains such as bioinformatics, cyber-physical systems, and social networks. As these applications encounter exponential growth in problem size due to the increased ability to generate, collect, and store relational data, being able to evaluate an architecture's suitability for data-intensive computing has become a pressing need. Network analysis plays a large and critical role in many of these data-intensive application domains, a fact recognized by recent efforts to design benchmarks with kernels based on graph algorithms such as SSCA2 [11] and Graph500 [10]. Unfortunately, real-world input graphs for testing are frequently unavailable, since the data may be proprietary or impossible to collect. This necessitates the ability to create synthetic networks which are "similar" to real data with respect to salient features, which can have subtle and surprising impacts on analysis. For example, the Graph500 benchmark originally used the R-MAT generator [20] to create its random instances, which was shown to significantly impact the expected performance of the breadth-first-search kernel due to large numbers of isolated vertices [78], and a multinomial degree distribution [38]. Efficiently generating "realistic" graphs at all scale sizes is a hard open problem; many generators exist, but a comprehensive comparison and associated tool for invoking them is currently unavailable. This led to the desire for an open infrastructure for generating and testing network data, as is described in this report.

In order to identify computer architectures which are best suited for massive complex network analysis tasks, we believe one must be able to (1) generate synthetic graph data at all scales, with well-understood and user-tunable network characteristics, and (2) emulate the run-time behavior of analysis algorithms made up of common key kernels, for example graph search or spectrum computation. In this report, we describe recent work on developing such an infrastructure for (1), including comprehensive testing of existing synthetic graph generator implementations for scalability and the integration of algorithms

calculating appropriate graph features/statistics into an open source framework.

2. BENCHMARKING: BACKGROUND & RELATED WORK

Computers vary widely in their architecture, core components, and capabilities. In order to be able to make a comparison between systems, one must identify a set of one or more objective measures of performance. Such metrics are especially critical to the High Performance Computing (HPC) community, where they are used as input to – and often requirements of – the procurement process. These performance measures, together with a prescribed process for generating them, are commonly referred to as benchmarks.

Over time, a large number of benchmarks have been proposed in the HPC community, some embraced, others not as much. In the remainder of this section, we describe several of the most prominent such benchmarks, and provide a summary of their history and current status.

We begin with three benchmarks which each focus on a single aspect of interest - Top500, Graph500, and Green500. It is worth noting that, as mentioned by Johnson [48], a better holistic approach might be a conglomeration of the three. However, he points out that this is extremely difficult, since not all systems submitting to the Top500 list also submit to the Graph500 list, and vice versa. In fact, in 2012 there were only about 120 supercomputers on the Graph500. To this end, we also include descriptions of three benchmarks which attempt to provide a standard which accounts for more than one aspect of interest in high performance computing: HPCC, SystemBurn, and HPCG.

2.1 HPL/TOP500

The Top500 list was started in 1993 by Hans Meuer at the University of Mannheim [2]. Published twice a year, for some this is the "gold standard" used to bestow the coveted title of most powerful supercomputer. Systems are benchmarked using Linpack (or High Performance Linpack (HPL)), with the fastest 500 maintained as part of the list.

Performance is measured by having the computer solve a system of dense linear equations, then reporting the largest problem solved and the number of FLOPS (floating point operations per second) [53, 68]. The benchmark has strict requirements on the algorithms and accuracy/precision used in computing the solution, and the advantage that the number of FLOPS (or, more recently, petaflops) is a single reportable number which is easily comparable across systems. Several researchers have put forth arguments against the Top500 benchmark, advocating that a single number is not a good representation of the overall system performance. In [23], the author details the seven primary algorithms used in high-end science on HPC platforms. Although the Linpack benchmark may be a good indication of system peak performance, dense linear algebra is only a small part of the overall makeup of an HPC application. As shown in Table 6 of [8], applications frequently use many of these kernels, and some have no dense linear algebra at all. We also observe the decision by NCSA (the HPC center overseeing the Blue Waters machine) to no longer submit results for the Top500 benchmark. Several important reasons given include the lack of relationship between the rankings and the system's usability by researchers, the fact that this race to the top encourages organizations to make poor choices when designing a new system, and lastly that what is actually being measured is the amount of funding organizations can secure for a new system [53]. One is left to conclude that Top500/HPL may not be an adequate solution for benchmarking HPC architectures going forward.

2.2 GRAPH500

Among others, Bader et al. [1, 10] realized that the problems represented by the Top500 benchmark were 3D-physics-type simulations, yet analytics (and specifically graph analytics) were becoming a

significant fraction of large computation. To address this, they proposed and developed the Graph500 benchmark as a complement to the Top500 in 2010, with a list that is also revised twice annually.

In the Graph500, FLOPS are replaced by a primary performance metric of TEPS (the number of traversed edges per second) during breadth-first searches (BFS, v.1.1) and single shortest paths searches (SSSP, v.2.0) in a large undirected graph. These algorithms stress the communications subsystem of a machine instead of measuring the speed with which it can perform double precision floating point operations. The problem instances here are random graphs constructed using a kernel based on the R-MAT model [20] or a Stochastic Kronecker Graph (SKG) model.

Unfortunately, this benchmark has several notable shortcomings that have led to a mixed review within the community. One common complaint was about the single method of generating input data, based on the R-MAT model. We note that the current implementation, Graph500 does offer an SKG generator as the default, as well as R-MAT. In addition, it has been shown that the reference implementations are actually not scalable in a large distributed environment [85]. Others argue that the benchmark is too limited to represent a "typical" graph workload (it has only two kernels, with SSSP just having been added in 2012), and does not encourage vendor innovation [1]. Lastly, results are accepted by the committee in a predefined format, but are not validated and (unlike Top500) site-specific implementations are not subject to peer review.

2.3 GREEN500

The third related benchmark is the Green500, which strives to measure the energy efficiency of supercomputers [34] while running the LINPACK/HPL benchmark. Conceptualized in 2005, and formally introduced in 2007, the Green500 list aims to raise awareness of the environmental impact of supercomputing. It is important to note that the Green500 is not a separate benchmark algorithm in itself - it simply relates the power usage of a supercomputer while running the HPL benchmark, resulting in a FLOPS/watt score. It is worth noting that the Green500 is still searching for the best way to quantify and report the energy efficiency using a single, easily comparable number. "The Green Index" (TGI) is the currently proposed metric [82], seeking to supplant the FLOPS per watt metric.

2.4 HPCC

Introduced in 2003, the High Performance Computing Challenge (HPCC) benchmark is a suite of seven tests: HPL, DGEMM, STREAM, PTRANS, RandomAccess, FFT and communication bandwidth & latency [28]. HPCC was designed to include more memory intensive tests which augmented the traditional HPL benchmark [64], and its authors claim this is a better indicator of overall system performance since it captures processor, memory and interconnect performance characteristics. We previously discussed HPL and RandomAccess. DGEMM measures the floating point performance of matrix multiplication, while STREAM measures sustained memory bandwidth. PTRANS, on the other hand, looks at communications while executing a parallel matrix transpose, and the FTT tests measures floating point performance while performing a 1-D Fourier Transform.

Unfortunately, the HPCC benchmark has failed to gain traction in the HPC community. Additionally, it has been criticized for only containing discrete workloads, yet not addressing integer-centric computations.

2.5 SYSTEMBURN

The SystemBurn benchmark, developed in 2012 at Oak Ridge National Laboratory, is an attempt to measure a more comprehensive set of a system's hardware features under maximal load [62]. It comprises configurable testing, but no discrete kernels or inputs are currently supported. Nineteen modules are included, several of which are also found in HPCC (e.g. DGEMM). Users have the ability to easily create and include new modules. However, SystemBurn was originally developed to create maximal load on a system, and not necessarily to measure performance, and its measurement capabilities are incomplete.

2.6 HPCG

The recently proposed HPC Preconditioned Conjugate Gradient [45] (HPCG) benchmark can be seen as an effort to produce a more relevant performance metric for current HPC applications. Rather than solving a dense linear system, HPCG computes a conjugate gradient on a large sparse matrix. As currently proposed, HPCG addresses a wider range of problems and computational kernels than HPL, but still fails to address issues such as integer-only calculations. Additionally, the options available for generation of the test problems (sparse matrices) is still unclear, and it appears to be constrained to storing the matrix in Compressed Sparse Row format. Since there is no available implementation yet, it is difficult to make a judgement on its real-world relevance and community impact.

3. RANDOM GRAPH MODELS

A graph is a mathematical object used to represent a set of objects (vertices) and the pairwise relationships among them (edges). For example, one might consider the set of students in a school as the vertices, and create edges between pairs of students who are friends. More formally, a **graph** $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subseteq V \times V$. Different types of graphs may have additional restrictions or information associated with them – for example, if E is a multi-set, then G is called a **multigraph**; alternatively, **simple graphs** forbid repeated entries in E and self-loops (edges of the form (u, u)). Other common variations include **directed graphs**, where the pairs in E are ordered, and **weighted graphs**, where members of V and/or E may have numerical weights associated with them.

The idea of creating and evaluating "random" graphs dates back to the late 1950s, when mathematicians defined several models [32, 37]. Formally, a random graph can be defined as a probability space over the set of graphs on a vertex set (of size n) determined by $\mathbb{P}[e_{i,j} \in G] = p$, where $0 \leq p \leq 1$ [81]. There is a significant body of literature evaluating the behavior of random graphs, including results on necessary and sufficient conditions for the formation of a giant component, expected degree distribution, and many other properties. Work on random graph models burgeoned as application communities became interested in generating synthetic graphs that were "realistic" (matched certain desired properties of their datasets). Some cited desired characteristics displayed by real graphs include heavy tail distributions for degrees and eigenvalues/eigenvectors, small diameters, edge densification and diameters that shrink over time [57]. We discuss graph features at length in Section 4.. It is also worth noting that more recent work on generators has emphasized their scalability, or ability to create massive graphs efficiently, a property that is especially important to the HPC benchmarking community, which requires very large problem instances to stress supercomputing architectures.

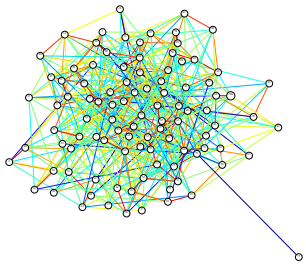
Through a search of the literature, the on-line community, and personal communications with researchers, a list of commonly-used models was assembled. We chose to restrict the scope to those models that had an existing and available implementation. Many current model implementations are limited in their ability to synthesize large scale graphs, and are not easily adapted for inclusion within a benchmark suite. Input parameters and underlying algorithms all contribute to controlling graph output properties and have been the source of much research – as illustrated by the variations in models. For additional background on previous work in graph generation, the most recent survey on the topic can be found in [19].

Our review of the models revealed a semi-natural grouping into four major classes: classic models (random graphs), Internet topologies, geometric models, and other real-world models, each of which is enumerated in this section. For a comprehensive list of models (and packages) explicitly studied as part of this report, see Table 3. A comprehensive description of all models can be found in Appendix A. Finally, in Table 1 we detail the runtime complexities of the studied models, as well as the potential for parallelization based on our understanding of the underlying algorithms. Where possible, we present references for formal proofs of these complexities; otherwise, these complexities are the result of approximations by the authors.

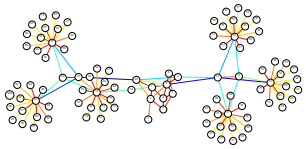
3.1 CLASSIC MODELS

We define classic models to be those random graph models which were conceptualized for the general study of random graphs, in the context of graph theory and theoretical computer science, not modern complex network analysis. These models typically have simplistic generative processes (such as a coin-flip to establish an edge between two vertices), and have a rich body of literature analyzing their average and limiting behavior as one varies the associated parameter(s).

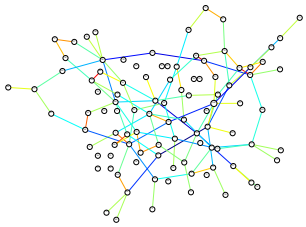
Erdös and Rényi are often cited as the pioneers of random graph generation models [32]; Gilbert's



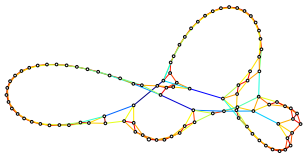
(a) Classical (ER, $p = 0.1$)



(b) Internet (Tiers, 2 MANS, 3 LANS)



(c) Geometric (Waxman, $L = .6, \alpha = .2, \beta = .1$)



(d) Real World (Small world, $r = 4, p = .02$)

Fig. 1. Examples of 4 model classes, 100 vertices, using a spring force directed layout algorithm

early work in the area was also influential [37]. The Ërdos-Rényi random graph model (often abbreviated ER, or referred to as the classical model) constructs a random graph by fixing a probability $p \in [0, 1]$ and probabilistically creating edges between pairs of vertices using independent Bernoulli trials. This well-studied model generates graphs whose degrees follow a Gaussian distribution, which is far from the heavy-tailed behavior seen in many real-world networks. One model developed to overcome this limitation while maintaining the simplicity necessary for rigorous mathematical analysis is the configuration model, which allows the user to specify the desired degree sequence. Each vertex is assigned a degree d_i, and thus d_i associated "half-edges". These "half-edges" are then paired randomly to create the final graph (the requested sequence of degrees must be realizable). Chung and Lu proposed a slight modification where an expected degree (weight) is assigned to each vertex, and similar to Erdös-Rényi, edges are generated using independent Bernoulli trials, but with probabilities now based on the ratio of the product of the endpoints' weights to the sum of all weights (see Appendix A.1 for more details).

This survey includes the following classic models: Erdos-Renyi [37], the configuration model [89], and partial k-trees [50] (which are designed to control a structural graph parameter called treewidth).

3.2 INTERNET TOPOLOGIES

The Internet is a prime example of a network topology with many vertices and edges (links). While certainly Internet topologies are a subset of real-world graphs, there is an extensive amount of work that focuses solely on generating networks with the specific properties of Internet topologies. Within this category, some have suggested a further breakdown of models based on the main properties they try to replicate (specifically random topology, hierarchy, and degree-related properties), though we do not draw such distinctions here.

It is worth noting that there has been some discussion as to the deficiency of these models [43] based on the discovery by Falutsos et al. [33] that the Internet-level Autonomous Systems (AS) topology has a degree distribution which follows a power-law, not replicated by many models in this category. Several

alternative models (for example, Barabasi-Albert, which we include in real-world models) have been created post- [33] publication.

This survey includes the following internet topology models: inet [47], Tiers [27] and Transit-stub [18]. All of these were developed to capture the hierarchical nature of the Internet [84].

3.3 GEOMETRIC MODELS

Here we describe a set of generative models which assume an underlying geometry influences the creation of edges between vertices. Although many internet topology models assume an inherent geometry in creating the vertex layout, we restrict our attention here to those which are not specialized to a particular application. For example, Young and Scheinerman proposed Random Dot Product Graphs (RDPG), which is a simple, fairly scalable model that associates a random vector in $\mathbb{R}^m$ with each vertex, and determines edges based on the inner product of the endpoints' vectors [91]. Unfortunately, there is not much empirical work comparing the outputs from this model with real-world data. A more recent model introduced by Krioukov et al [54] builds on the observation that many real-world networks are better visualized using embeddings in hyperbolic space (instead of Euclidean).

This survey includes the following geometric models: Waxman [87] (which some identify as a precursor to Internet models), Random Dot Product Graphs (RDPG) [76, 91], and the Krioukov hyperbolic generator [54].

3.4 REAL-WORLD MODELS

The world around us contains constantly evolving networks such as social interactions, road connections, or even corporate networks. Models that fall into this class were explicitly designed to mimic one or more properties of networks that has been empirically observed in real-world data. Three major concerns with classical models were the lack of a natural notion of network growth, non heavy-tailed degree distributions, and low clustering coefficients.

To address the need for higher clustering coefficients, Watts and Strogatz introduced the Small World model [86]. At a very high level, Watts-Strogatz small world graphs are created by starting with a regular ring lattice on the prescribed number of vertices, then "re-wiring" a certain number of edges with their new endpoints depending on a probability β and the distance in the original lattice. The resulting graphs have a short average path length and high clustering coefficients. Unfortunately, this model tends to create unrealistic degree distributions and forces a fixed number of vertices, which inhibits its use to model network growth.

Although the configuration model technically addressed the need for generating graphs with a power-law degree distribution, many researchers were unsatisfied, since it was not based on a natural notion of network growth over time (and thus could say nothing about the mechanisms likely responsible for formation of heavy-tailed degree distributions). The notion of preferential attachment (new vertices are more likely to connect to other high-degree vertices) inspired a number of generative models, including one of Barabási and Albert, denoted BA, which many refer to as generating scale free graphs [13]. However, the graphs generated with the BA do not have the high levels of clustering seen in real networks, have a fixed power law exponent [5], and have an exponential cutoff. Several authors, including Barabási and Albert themselves, have suggested modifications to the initial scale free model, primarily to the linear growth and linear preferential attachment components [5, 71].

More recently, an interest in the scalability of random graph models has prompted a new wave of generators, including R-MAT/SKG, RDPG, and BTER. Citing the inability of previous models to match

"real-world" degree distributions (which are typically unimodal and exhibit a power-law), to exhibit "community" structures, and satisfy criteria such as a small diameter or certain eigenvalue distribution, Chakrabarti et al. introduced the R-MAT model [20]. Widely adopted, the highly parallel model uses four probability parameters recursively to rapidly dictate edge creation. Leskovec et al. proposed a generalization of R-MAT named the Stochastic Kronecker Graph model (SKG) [57, 59] which uses the Kronecker product. The SKG model is easily parallelizable, requires few inputs, and quickly generates graphs. SKG is the current default kernel in the Graph500 benchmark, while R-MAT variants are still available as options.

More recently, Seshadri et al. have proposed the Block Two-Level Erdös-Rényi (BTER) model [52, 77] which integrates ideas from both the ER and Chung-Lu configuration models. The authors report results of a generated graph with over 4.6 billion edges. The primary motivation behind their work was to maintain scalability while improving the reproduction of realistic community structure. Finally, we mention an approach rooted in multiscale methods – the MUSKETEER model [40] was recently introduced in an effort to create networks closely representing reality with respect to structure and at multiple scales. A diversity of other models continue to be introduced in an effort to address the needs of various communities or improve upon existing models (see for example, variations on the preferential attachment model which include the "forest fire" model [60] and "winner does not take all" [71]).

This survey includes the following "real-world" models: Exponential Random Graph Model (ERGM) [46], MUSKETEER [40], preferential attachment [80], small world models [86], and Stochastic Kronecker/R-MAT [20, 57].

Table 1. Model complexities and scalability

Model	Complexity	Ref.	Scalability/ Parallelization
BTER	$O(\|E\| \log d_{max})$ where d_{max} is the maximum degree of the input distribution	[52]	[b]
Configuration model	$O(\|E\|)$	[a]	[c]
Ërdos-Rényi	$O(\|V\|^2)$	[66]	[d]
Ërdos-Rényi (fast)	$O(\|V\| + \|E\|)$	[14]	[e]
Inet	$O(\|V\|\|E\|)$	[a]	[c]
Krioukov	$O(\|V\|^2)$	[54]	[f]
Kronecker	$O(\|E\|)$	[57]	[b]
Musketeer	$O(\|E\|)$	[40]	[f]
Partial k-trees	$O(\|V\| + \|E\|)$	[a]	[f]
Preferential attachment	$O(\|V\|^2)$	[14]	[g]
Preferential attachment (with sampling)	$O(\|V\| + \|E\|)$	[14]	[g]
RPDG	$O(\|V\| + \|E\|)$ for sparse graphs	[a]	[f]
R-Mat	$O(\|E\| \log \|V\| \log \|E\|)$	[20]	[b]
Small-world	$O(k\|V\|)$ where k is neighborhood size	[a]	[f]
Tiers	$O(\|V\|^2)$	[27]	[f]
Transit-stub	$O(\|V\|^2)$, based on construction of Ërdos-Rényi or Waxman graphs	[a]	[f]
Waxman	$O(\|V\|^2)$	[27]	[f]

[a] complexity determined by inspection of algorithm

[b] highly parallel implementation exists

[c] dependence on current graph state makes parallelization difficult

[d] parallel implementation should be trivial

[e] dependence on current algorithm state makes parallelization difficult

[f] parallel implementation should be possible but perhaps non-trivial

[g] not as currently specified - possibility exists for modification of algorithm

4. GRAPH FEATURES

In order to compare graph generators, one must define a set of measures on the algorithm/implementation and its output. In this section, we focus on how one might evaluate the output, typically by analyzing and comparing some set of features/statistics of the resulting graph.

While no report currently exists with a comprehensive comparison of all random graph generators, various metrics are used in the evaluations that do exist, and no real consensus exists on which graph properties are most important/salient/appropriate for evaluating how realistic a synthetic graph is. In our work, we include a broad spectrum of graph features which have been used in one or more prior comparisons. This section first describes each feature for which we have an implementation (along with some intuition as to why it is relevant for evaluating random graphs), then gives the computational and storage complexities, and concludes with a discussion of how these features can impact graph analysis algorithms.

Rather than integrating multiple specialized graph analysis codes, all features listed in Section 4.1 have been implemented in the INDDGO framework detailed in Appendix B. INDDGO provides a much more straightforward interface than other libraries such as the Boost Graph Library, which enabled us to rapidly design and implement the desired feature calculations. Additionally, this gives us complete control over the statistics output. We also incorporated OpenMP parallelism where appropriate, as we intend to perform the majority of our analysis on large shared-memory machines. While the use of INDDGO this may affect the speed of some calculations, we believe the ease of use provided by using a single package outweighs the potential losses. Likewise, the use of OpenMP should give us an advantage over many of the existing serial codes.

4.1 FEATURE DESCRIPTIONS

Unless otherwise noted in the feature descriptions, all calculations are performed on both the entire graph (which may have multiple components), and on its largest connected component. Both results are reported back to the user.

- **All Pairs Shortest Paths (APSP)** [94] is an algorithm which calculates a shortest path between all pairs in the graph. We also use the term to refer to the graph feature consisting of the pairwise distance matrix obtained from this calculation. Practically, this may be computed by running a single source shortest path **(SSSP)** [94] algorithm, which finds the shortest paths (minimum edge weight sum in a weighted network) from a given vertex (the source s) to all other vertices in the graph. Both of these are often used in the computation of other metrics (such as eccentricity), but are also useful independently (e.g. in determining optimal network routing).
- A graph's **average degree** [65] is the ratio of the number of edges to the number of vertices in the graph. This metric implies other statistics, such as the minimal degree of a subgraph. Additionally, it is used in evaluating things like the equilibrium of a graph (how the degree distribution affects the density) [73].
- The **average path length** [90] of a graph is the arithmetic mean of all $\binom{n}{2}$ shortest path distances (usually computed using All Pairs Shortest Paths), and is an indicator of the efficiency with which information flows through a network. When a graph has multiple connected components, we average only the finite lengths.
- **Betweenness centrality** [65] is a popular measure of the "importance" of nodes in a network, relative to the flow of information along shortest paths (as is common in routing and other

applications). The centrality is calculated as the ratio of the number of shortest paths which pass through a given node to the total number of shortest paths in the network. It is determined by calculating the fraction of shortest paths passing through a given node to all shortest paths. This metric is computationally intensive, and numerous algorithms and implementations exist (see, for example, [12]).

- The **clustering size** [67] or **coefficient** [58, 65] deals with the **number of triangles** in a graph. The *global clustering coefficient* of a graph is the number of "closed" triplets (subsets of 3 fully-connected vertices) in a graph divided by the number of possible triplets ($\binom{n}{3}$). A related measure, the *local clustering coefficient*, is computed for each vertex, and measures how close the neighbors of that vertex are to being a complete graph. Clustering metrics can give insight into properties of communities within a graph.
- Since graphs generated may not be connected, we calculate a sorted list of the **component sizes** (number of vertices in each connected component). For connected graphs, this will be a single number (equal to the *graph size*, below).
- The **degeneracy** [61] of the graph is the maximum *k-shell number* among the vertices in the graph. As we discuss in section 4.2, this can be useful in bounding the complexity of other graph algorithms.
- A graph's **degree assortativity** [69, 70] measures the tendency of vertices of degree d to connect to other vertices of similar degrees. To calculate the degree assortativity of a graph, we iterate over every edge of the graph, xy. Let d_x be the excess degree of vertex x (i.e. $deg(x) - 1$), and define d_y likewise. We calculate 3 sums over all the edges: $n_1 = \sum_{xy} \frac{d_x \cdot d_y}{m}$, $n_2 = \sum_{xy} \frac{d_x + d_y}{2m}$, and $d_e = \sum_{xy} \frac{d_x^2 + d_y^2}{2m}$. The degree assortativity is then $r = \frac{n_1 - n_2}{d_e - n_2}$. In an Internet network, degree assortativity indicates the tendency for highly-connected vertices (routers) to connect to other high degree vertices (a similar phenomenon is observed among vertices with very few connections (servers)). Assortative mixing is often studied in social network analysis and epidemiology as discussed in [70].
- The **degree distribution** [58, 65] describes the connections of the vertex degrees of the graph. We define Δ to be the maximum vertex degree. For each integer $0 \leq d \leq \Delta$, the number of vertices with degree d is computed, and a probability distribution is fit to the results. See [21] for details on power-law fitting. Since many real-world networks are known to have heavy-tailed or power-law degree distributions, this is one of the most commonly used graph features for determining how well a model approximates realistic data.
- The **δ-hyperbolicity** of a graph [22] provides a measure of how well distances are preserved if a graph is embedded into a hyperbolic space. More specifically, δ is large if there is significant distortion in shortest-path distances between vertices when they are approximated by distances along the "best-fit" tree. When δ is zero, all shortest paths are unique, and no distortion occurs (this is true in trees and complete graphs, for example). On the other hand, $n \times n$ grids are $n/2$-hyperbolic (intuitively because there are many different shortest paths between pairs of vertices). This measure is only computed on the largest component, since its calculation involves distances between sets of four points, and is not meaningful when some of those values are infinite (for vertices in different connected components).
- A graph's **diameter** [56] is the maximum shortest path distance between any pair of vertices (we define the distance between vertices which are not connected to be infinity). This measure is susceptible to outlier values.

- The **edge density** [67] is the ratio of the number of edges ($|E|$) to the maximum number of possible edges ($\binom{n}{2}$ for a simple, undirected graph). This measure of sparsity is commonly used to help determine appropriate storage formats (compressed sparse row, dense adjacency matrix, edge list, etc.).
- Similar to *diameter*, the **effective diameter** [19, 58] is the minimum number of hops (shortest-path distance) in which some percentage of all connected pairs of vertices can reach each other. In the literature, 90% is a commonly agreed-upon threshold, and is used in this study. This measure was developed as an alternative to *diameter* that is not as susceptible to outliers.
- The **eigenvalues** or **eigenvalue spectrum** [44, 65, 84] (often referred to simply as the **spectrum**) are the eigenvalues of the 0/1 adjacency matrix. Due to computational complexity, we restrict ourselves to computing the three largest and three smallest eigenvalues (currently using the SLEPc [44] library). We note the solver may return fewer results depending on the size of the matrix and the maximum number of allowed iterations.
- The **expansion** [84] or **distance distribution (normalized expansion)** [65] of a graph measures the average fraction of vertices at distance at most h from a given vertex. For each integer $1 \leq h \leq diameter(G)$, the number of vertices reachable with paths of length at most h is computed for every vertex in the graph, then averaged (and possibly normalized). The distribution reported is the averages over the range of h. Alternate definitions may refer to a "ball of radius h centered at node x". This metric reflects how rapidly neighborhood sizes grow as you expand from a vertex, and is important in evaluating the performance of routing algorithms and the potential impact of viruses/worms in a computer network.
- The **graph size** [67] is the number of vertices in the graph, and is the usual measure of "scale."
- The **k-core** [61] of a graph is the maximal subgraph where all vertices have degree at least *k in the subgraph*. A vertex has **k-shell number** κ if it is in the κ-core, but not in the $(\kappa + 1)$-core. This metric can be useful in evaluating community structure and hierarchy in networks.
- The **node diameter distribution** [84] provides a distribution of the lengths of the longest shortest path from each vertex (the **eccentricity**). Practically, this is computed as the maximum observed at each vertex during the All Pairs Shortest Paths (APSP) computation. This metric can be usefuli in determining the worst-case scenario in routing networks, and is closely related to the *effective diameter*.

4.2 FEATURE IMPACT

Although we seek to understand the behavior of these features in order to better emulate real-world graphs with synthetic data, many of these characteristics also have an impact on the performance (complexity, average case running time, or memory usage) of important graph algorithms. The complexity can be expressed in terms of $|V|$ and $|E|$, and in some cases in terms of their ratio (the edge density). For example, the best known bound for calculating betweenness centrality is $O(|V| \cdot |E|)$. See Table 2 for additional complexity bounds.

One way in which features may impact algorithmic efficiency is through fixed parameter tractability (FPT) [36]. That is, when a specific property of the graph has bounded value, there are polynomial time algorithms for solving otherwise NP-hard problems. For example, the problem of finding a dominating set [29] is NP-complete, but has been shown to have FPT algorithms (and even polynomial kernels) in graphs of bounded degeneracy [6, 72]. Similarly, the classic NP-hard problem of graph isomorphism [29]

can be solved in polynomial time when bounding the multiplicity of eigenvalues [9] or the maximum degree [63].

In the cases above, special algorithms have been designed to reduce worst-case complexity by taking advantage of the graph structure implied by the bound on a graph feature. In other situations, properties of the graph can have significant effect on the typical running time of an algorithm (but not a theoretical impact on the worst-case complexity). One common example of this occurs in parallel breadth-first-search algorithms (BFS), whose performance can depend heavily on degree distribution. There have been many papers providing empirical reports on the effect of a skewed degree distribution, and suggesting alternative algorithms which are specialized to handle graphs with this characteristic (for example, see [79] and [16]).

Table 2. Feature complexities[a]

Algorithm name	Best known bound (dense)	Best known bound (sparse)	Implemented bound	Best storage bound	Implemented storage bound	Reference
All pairs shortest paths (APSP)	$O(\|V\|^3)$	$O(\|V\| \cdot \|E\|)^b$	$O(\|V\|^3)$		$O(\|V\|^2)$	[74]
Average degree[d]	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	[65]
Average path length		$O(\|V\| \cdot \|E\|)$	$O(\|V\|^2)$ [e]		$O(1)$ [e]	[90]
Betweeness centrality	$\Theta(\|V\| \cdot \|E\|)^b$	$\Theta(\|V\| \cdot \|E\|)^b$	$O(\|V\| \cdot \|E\|)$	$\Theta(\|V\| + \|E\|)$	$O(\|V\| \cdot \|E\|)$	[17]
Clustering coeff. (triangle counting)	$O(\|V\|^{2.376})$	$\Theta(\|E\|^{3/2})$ or $O(\|V\|^{2.376})$	$\Theta(\|E\|^{3/2})$	$\Theta(\|V\|)$	$\Theta(\|E\|)$	[55] [7]
Degeneracy	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$		$O(\|V\|)$	[15]
Degree assortativity			$O(\|E\|)$		$O(1)$ given degrees	[69]
Degree distribution	$O(\|V\|)$		$O(\|V\|)$		$O(\|V\|)$	[65]
δ-hyperbolicity	$O(\|V\|^{3.69})$ or better	$O(\|V\|^{3.69})$ or better	$O(\|V\|^4)$		$O(\|V\|^2 + \|E\|^2)$	[35] [22]
Diameter	$O(\|V\|^3)$		$O(\|V\|^2)$ [e]		$O(1)$ [e]	[56]
Eccentricity			$O(\|V\|^2)$ [e]		$O(\|V\|)$ [e]	[83]
Edge density [e]	$O(1)$	$O(1)$	$O(1)$		$O(1)$	[67]
Effective diameter	$O(\|V\|^3)$		$O(\|V\|^2)$ [e]		$O(\|V\|)$	[58]
Eigenvalues			$O(\|V\|^2)$		$O(\|V\|^2)$	
Expansion	$O(\sqrt{\log \|V\|})^c$		$O(\|V\|^2)$ [e]		$O(\|V\|)$	[75]
Graph size [d]	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	
K-cores	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$		$O(\|V\|)$	[15]
Node diameter (eccentricity) distribution	$O(\|V\| \cdot \|E\|)$		$O(\|V\|^2)$ [e]		$O(\|V\|)$ [e]	[83]

[a] for exact algorithms, undirected graphs

[b] unweighted graphs

[c] approximation for the sparsest cut; Cheeger's inequality

[d] algorithm depends on values already maintained for all graphs, thus can be calculated in constant time

[e] given APSP matrix, otherwise add $O(|V|^3)$ computational and $O(|V|^2)$ storage complexity

5. INFRASTRUCTURE FOR INSTALLATION AND TESTING

5.1 BUILD AND INSTALLATION FRAMEWORK

Implementations of all generators detailed in Appendix B were downloaded and stored in a centralized `git` repository. The choice to keep the generators in a revision-controlled repository allows us to see what, if any, changes were required to make the generators function in our environment. In addition, a `README` file is maintained for each generator, detailing reasons for the changes. These were kept to a minimum necessary to work within our testing framework. This included adding additional output formats, adding timing routines, and, for many of the generators written in Matlab, Python, and R, writing short wrapper scripts to call the libraries.

To automate the configuration/build/install process with consistent naming conventions across all generators, custom scripts were written for each package. This allows us to easily write a wrapper that rebuilds each generator appropriately, and extend to include additional packages in the future. These scripts are often a simple "`make ; make install`," however some generators require more steps, and capturing these steps in a script creates an interface which does not require the user to manually build the package, ensuring consistency and removing potential user implementation error.

As seen in Table 4, the tested graph generators are implemented in a wide variety of languages, using numerous libraries. One difficulty presented by the breadth of generators considered is the necessary comparison of compiled languages, such as C++, with interpreted languages like Python and R. The latter are often slower than their compiled counterparts. Furthermore, just as compiler choice can affect runtime, the choice of interpreters can drastically influence the runtime characteristics of an algorithm, as shown in [3]. As a result, while runtime statistics are collected for all generators, they will not be used as one of the primary comparison metrics.

Another consideration is output format for the generated graphs; we standardized to use simple ASCII edge lists or adjacency lists, and modified generators that did not originally support these formats to enable them. Choosing just two plain, human-readable output formats simplifies the post-processing work required for calculating graph features, and standardizes the approximate amount of I/O required at a given scale.

5.2 TEST FRAMEWORK

After building and installing the graph generators, a repeatable framework was desired for testing purposes. Additionally, the ability to record and easily extract metadata from the testing runs was considered crucial in this study. Given these requirements, we designed a relatively simple Python template script that may be copied and modified when adding new generators. The testing script takes a configuration file as an argument, and generates graphs based on the parameters specified in the file. After the run, the script also writes a `.meta` file containing relevant metadata for each graph produced. The metadata includes the total runtime of the generator, the time the generator spent in actual generation versus the time spent writing the file to disk, memory usage, date and time of the run, as well as the exact command line invoked to produce the output graph.

We present an example of the script used to generate ER graphs using the APGL package in Listing 1. As input to this script, we pass a filename. The format of this file is seen in Listing 2. This input file will generate $7 \cdot 3 = 21$ different output graphs, one for each possible combination of vertices and probability. Along with the graphs, we get a metadata file that is show in Listing 3.

Additional scripts were written to aggregate output from multiple `.meta` files into a format easily

Listing 1. Python Wrapper Script

```
#!/usr/bin/env python

import sys
import re
import ornlgraphsurvey as og

# define these for your generator
command = "../generators/bin/er-apgl.py"
default_config = "../configs/er-apgl.default"
required_args = ['vertices', 'filename' ]

def main():
    if len(sys.argv) == 1:
        print "using default configuration"
        config_file = default_config
    elif len(sys.argv) == 2:
        config_file = sys.argv[1]
    else:
        print "Usage: "+sys.argv[0]+": <configfile>"
        sys.exit(0)

    args = og.parse_config(config_file)

    if not og.validate_args(args, required_args):
        print "Please check your configuration file for required arguments"
        sys.exit(1)

# change this for your particular generator
    for v in args['vertices']:
        for p in args['prob']:
            fname = args['filename'][0]+"_v"+str(v)+"_p"+str(p)
            fname = og.get_unique_filename(fname)
            meta = fname+".meta"
            cmd = command+" -v "+str(v)+" -e "+str(p)+" -f "+fname
            print "running "+cmd

# write required/common header info
            m = og.meta_init(meta, cmd)

# run the command and capture the output and memory usage
            (output, mem) = og.get_output_and_usage(cmd)

# write any other useful information
# we assume that the generator outputs generation and file write time
            m.write("Memory: "+str(mem)+"\n")
            m.write("Vertices: "+str(v)+"\n")
            m.write("Probability: "+str(p)+"\n")
            m.write(output)
            m.close()

if __name__ == "__main__":
    main()
```

Listing 2. Harness configuration file

```
# generate graphs at different power of 2 scales and probabilities
# save to filename starting with er-apgl
vertices 1024 2048 4096 8192 16384 65536 262144
prob 0.001 0.01 0.05
filename /data/graph/er-apgl
```

Listing 3. Output metadata

```
Command: ../bin/er-apgl.py -v 1024 -e 0.001 -f /data/graph/er-apgl_v1024_p0.001
Hostname: xxxx01.ccs.ornl.gov
Time: 2013-08-26-14:32:39
Memory: 44016
Vertices: 1024
Probability: 0.001
Generation time: 0.250616788864
Output time: 0.0188760757446
Total time: 0.269492864609
Output edges: 500
Output vertices: 1024
Output format: edge
Output filename: /data/graph/er-apgl_v1024_p0.001
```

parsed by common data analysis packages such as GNUPLOT. In the next phase, we intend to create plots of features such as the degree distribution. Additionally, for smaller graphs, we may utilize INDDGO [39] routines in conjunction with GRAPHVIZ [30] to visualize the graphs themselves.

5.3 INITIAL EVALUATION RESULTS

After installation and verification of the various generators' ability to run, a few basic tests to gauge any initial limitations were performed. After this first round, the main restrictions appear due to slow serial algorithms, memory (the generator uses all of the computer memory during generation), and the implementation language (e.g., Matlab, R). Another challenge is determining whether poor generator performance is due to the algorithm itself or an inefficient implementation.

For the scope of this report, the acceptable runtime limit is set to 24 hours per individual graph. Thus, while some generators can surely generate graphs larger than analyzed here, the results will not be included as part of this study. It is possible that future optimized versions of these generators (e.g. in C/C++ or even parallel implementations) and their resulting graphs will be used. A list of the packages and models implemented, as well as the initial limitations discovered, can be found in Table 3.

Table 3. List of generators and limitations

Generator	Package	Limit	Limitation
Configuration model	APGL	~ 37 million vertices	[a]
Configuration model	NetworkX	~ 37 million vertices	[a]
Erdos-Renyi	APGL	2^{18} vertices	time
Erdos-Renyi (block 2-level)	BTER	2^{28} vertices	memory
Erdos-Renyi (gnp method)	GGEN	~ 2^{18} vertices	memory
Erdos-Renyi (layer method)	GGEN	~ 2^{18} vertices	memory
Erdos-Renyi	NetworkX	depends on parameters	time
Erdos-Renyi (fast variant)	NetworkX	ran @ 2^{22} vertices, not @ 2^{24}	time
ERGM	ergm	$\sqrt{\frac{2^{32}-1}{2}}$ vertices	Int_max = $\frac{2^{32}-1}{2}$
Hyperbolic	Krioukov	2^{20} vertices	time
Inet	inet-3.0	$3037 < n \leq 2^{17}$	time, model limitation
Kronecker (R-Mat)	pywebgraph	2^{20} vertices	time
Kronecker (R-Mat+noise)	Graph500	2^{30} vertices	memory
Musketeer	Musketeer	2^{18} vertices with 8% and 7% edit rates	time
Partial k-tree	INDDGO	2^{27} vertices	memory
PLRG	Boost	2^{30} vertices	memory
Preferential Attachment (B-A)	APGL	2^{16} vertices	memory and/or time
Preferential Attachment (B-A)	NetworkX	2^{23} vertices	time
RDPG	MFR	<46000 vertices	memory
Small World (Watts-Strogatz)	APGL	at least 2^{26} vertices	time
Small World (Watts-Strogatz)	NetworkX	2^{24} vertices	time
Tiers	tiers	2^{26} vertices	memory
Transit-stub	gt-itm	~ 2^{18} vertices	unknown; gt-itm bug?
Waxman	stocksim	<180000 vertices	memory
Waxman	NetworkX	8000 vertices with default a=0.4, b=0.1	time

[a] 37 million vertices was largest degree distribution generated at the time of testing

Tests were run on a 48-core HP DL585 G7 with 384GB of RAM. Refer to Table 4 for details on package implementation languages, which may have an effect on the limitations.

6. CONCLUSIONS AND NEXT STEPS

The impetus for this study came from the desire to have a graph-centric benchmarking capability similar to that of SYSTEMBURN (designed for computational profiling under maximum load). The need to be able to generate synthetic graphs at larger scales is widely recognized, but the plethora of different model types may leave researchers in a decision quandary. In this report, we describe the implementation of an open framework for compiling and instantiating a broad spectrum of readily available generators (including those which required minor source code modification). Additionally, we report on initial testing of some common limitations, illustrating how some models and implementations may or may not be useful in a future benchmarking suite. To enable further testing and comparison, a comprehensive suite of graph features and statistics was compiled and implemented in a common open-source framework. With the elements described here in place, the testbed is set to run all identified generators with user-configurable parameters, compute and compare calculated features of the graphs created, and determine whether stated expectations (from existing literature) are met. We can also evaluate the behavior of both the generation and output statistics as the scale of graphs being generated is increased.

The initial set of gathered generators is large and might be considered unwieldy; we anticipate a continual down-selection of models and implementations based on testing results and sponsor input/community interest. The primary criteria for inclusion will be a generator's ability to match desired graph features and ability to create or identify a scalable implementation. Future work includes evaluating whether various generator codes can be improved through parallel implementation, as well as their behavior when instantiated on different computer architectures.

This report describes the necessary foundational work required to set up the generator test-suite. A follow-on report is anticipated describing the results from the down-selection, analysis, OPENSHMEM or UPC generator implementation and finally the integration with sparse benchmarks.

ACKNOWLEDGMENTS

This work was supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.

INDDGO development has also been supported by the DARPA GRAPHS program and the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research was supported by an allocation of advanced computing resources provided by the National Science Foundation. The computations were performed on Kraken and/or Nautilus at the National Institute for Computational Sciences (http://www.nics.tennessee.edu/).

Appendices

A DESCRIPTION OF MODELS

A.1 IMPLEMENTED MODELS

BTER [52,77] Proposed by Seshadri, Kolda and Pinar, the Block Two-Level Ërdos-Rényi (BTER) model seeks to better capture community structure.

The main algorithm operates in two phases with an initial pre-processing step. The model can handle any degree sequence d_i, which must be given as input. This also dictates $|V|$. In the pre-processing step, groups are created by first ordering the degrees with $d_1 \leq d_2 \leq ... \leq d_n$. Starting with the smallest degree, $d_i + 1$ vertices are removed from the list to form a community. The process is repeated until the list is empty. Within a given community, vertex degrees are as uniform as possible. In *Phase 1*, small independent communities C_i are generated following an ER model. Edges are created within each community with probability $p = f(\mathsf{log(smallest\ degree)}\ \mathsf{in}\ C_i)$. In *Phase 2*, the C_i are connected using a Chung-Lu model. For some set of the isolated vertices, edges are manually created. For the theory and further details, we refer the reader to the original description in [77].

Results from initial comparisons with the Chung-Lu model [77] indicate that the clustering coefficients of BTER graphs have a closer match to real data. The authors note that since these results in are more in tune with true community structure, other parameters such as the eigenvalues also have a better match to their actual expected values.

The current generator implementation is in Matlab, though the algorithm is parallelizable. The creators of BTER claim that the model is quite scalable, with tests up to 4.6 billion edges (using Hadoop) resulting in very realistic models.

Configuration model [70,89] A random graph model that creates a graph with an arbitrary degree sequence. Given a desired degree sequence $d_1 \leq ... \leq d_n$, we can construct a graph with vertices $v_1, ..., v_n$ where vertex i has degree d_i. Initially, each v_i is assigned d_i empty "slots" to which connections can be made. Two empty slots are chosen at random from the graph and an edge is created. This process continues until there are no more free slots in the graph. The resulting graph will have vertices with the exact specified distribution but may contain vertices with self-loops or multiple edges.

Chung and Lu proposed a slight modification where an expected degree (weight w_i) is assigned to each vertex, this sets the expected degree sequence as opposed to the actual sequence. Edges are then constructed through independent Bernouilli trials where the probability of forming edge e_{ij} is given by $p_{ij} = \frac{w_i w_j}{\sum_k w_k}$. This change provides a more natural extension of the Ërdos-Rényi model, which was presented as one disadvantage of the unmodified version.

Ërdos-Rényi [31,37] Ërdos-Rényi random graphs are synthetic graphs generated using one of two models first described in 1959.

These graphs are among the most straightforward synthetic graphs to construct. In [31], Ërdos and Rényi describe a method for choosing a random graph from among all possible graphs given n vertices and M edges. This is equivalent to the method in [37], in which a graph is constructed with n vertices and probability p of an edge between two given vertices. For each vertex $n_i \in |V|$, an edge is created between n_i and each other vertex with probability p.

Inet [47,88] The Inet generator is a purpose-built model intended to produce graphs that model the Internet and its growth at an Autonomous System (AS) level. By analyzing actual data collected about the Internet AS topology from 1997-2000, the authors are able to observe patterns in the growth of the network. Using this data, the claim is that Inet can generate these AS level topologies for time scales into the future.

As input, Inet takes two parameters: the number of vertices N, and k the fraction of vertices that have a degree of 1. Using the gathered data and resulting exponential growth parameters, the method calculates the time t that it would take the number of ASes on the Internet to grow to N. Using this information, it is possible to calculate a degree distribution for the given vertices (with some modifications based on the real-world observations).

Once a degree distribution is established, Inet creates a spanning tree among all vertices with degree > 1. Starting with an empty graph, G, a random vertex not in G is added to the graph. Once all vertices with degree > 1 are connected to G, the degree 1 vertices are then connected, at random, to existing vertices in G with unfilled outdegrees. Finally, any remaining unfilled outdegrees are connected, beginning with the largest degree vertex. This appears to intuitively mimic some of the network growth behavior observed when looking at the interconnection between ASes on the Internet.

Krioukov [54] The Krioukov graph generator draws on ideas from hyperbolic geometry. The authors contend that current simple random graph models do not accurately reproduce strong heterogeneity and clustering which are common properties of large networks. Using hyperbolic geometry, a new graph generator capable of integrating these properties is proposed.

Conceptually, the algorithm works by taking a compact region in a hyperbolic space. N vertices are placed within the space via a Poisson point process and edges are created between sets of vertices if the distance is less than a specified parameter (for example, R in the case of a circle with radius R). The claim is that this accurately replicates real vertex degree distributions. Further details about model parameters and cases are given in [54].

Kronecker [56] A generative model that uses the Kronecker product to create graphs (denoted as "Kronecker graphs"). This method was proposed to address shortcomings seen with other generators. The claim is that the Kronecker method can generate graphs with a power-law degree distribution and a small diameter. Further, it is claimed that the generator more closely models temporal effects seen in real-world networks, such as densification and a shrinking diameter over time [56].

Given a graph G with n vertices, we can generate a new graph using the Kronecker model by taking the Kronecker product (denoted $\otimes$) of G's adjacency matrix with itself, or $adj(G') = adj(G) \otimes adj(G)$. G' will have 2^n vertices.

MUSKETEER [40] Many times when generating graphs similar to real-world networks, there is not a readily available model that reproduces all of the features present in the real-world network. MUSKETEER is a multiscale graph generator that claims to solve this problem to some degree. The model takes an input graph, and given some parameters subsequently detailed, generates a new graph with features similar to the original graph.

Several transformations are available to generate new graphs from the original one such as: editing edges or vertices, and adding edges or adding vertices. Each of these edits may be applied at a different level of coarseness. MUSKETEER coarsens the graph a given number of times (specified

by the levels at which the edits are applied) by computing a projection of the Laplacian [40]. Once changes are made at the lowest level, the graph is un-coarsened by computing the reverse projection of the original vertices. In the case of new/edited vertices, vertices are copied from the original graph.

New vertices added at the coarsest levels are expanded during the uncoarsening process to multiple vertices. As a result, small changes at the deepest levels can have a large effect on the resulting graph. The resulting graph replica consists of the original graph, plus the results of inserting resampled portions of the original graph into the edit sites. Given this, the replica should have good fidelity with respect to features of the original.

Partial k-trees In computational graph theory, numerous *NP*-hard graph problems can be solved in polynomial time for graphs with bounded treewidth (a parameter measuring how "tree-like" the cut structure of a network is) [25]. Generating graphs with a known upper bound on its tree-width uniformly at random is possible using partial k-trees.

The class of *k-trees* is defined recursively. In the smallest case, a clique on $k + 1$ vertices is a k-tree. Otherwise, for $n > k$, a k-tree G on $n + 1$ vertices can be constructed from a k-tree H on n vertices by adding a new vertex v adjacent to some set of k vertices which form a clique in H. A k-tree has treewidth exactly k (the bags of the optimal tree decomposition are the cliques of size $k + 1$). The set of all subgraphs of k-trees is known as the *partial k-trees*.

It is easy to see that any partial k-tree has treewidth at most k (one can derive a valid tree decomposition of width k from that of the k-tree which contains it). Furthermore, any graph with treewidth at most k is the subgraph of some k-tree [50]. Thus the set of all graphs with treewidth at most k can be generated by finding all k-trees and their subgraphs, leading to an easy randomized generator for graphs of bounded treewidth.

Preferential Attachment [13] The Preferential Attachment model identified by Barabási and Albert in [13] posited that real-world networks often exhibit two behaviors for which previous models such as the Ërdos-Rényi model did not account: growth and a tendency for new vertices in the network to preferentially attach to existing vertices with higher degrees. This lead to the concept of a "scale-free" graph, or a graph whose degree distribution follows a power law.

To generate a graph using the Preferential Attachment model, one first seeds the graph with an initial set of vertices, m_0. Some generators [26] then assign each vertex an equal probability without creating any edges, while others [41] then create a clique from these. New vertices are added individually to the graph, connecting each one to $m \leq m_0$ others with probability p_i where $p_i = \frac{d_i}{|E|}$. As the network grows d_i and $|E|$ change. Vertices attach "preferentially" to existing high degree vertices.

RDPG [91] The Random Dot Product Graph (RDPG) was initially formalized by Kraetzl, Nickel, Scheinerman and Tucker and furthered by Young and Scheinerman. The extension models allow for both undirected and directed graphs and generalize Ërdos-Rényi and configuration models.

The basic algorithm operates as follows. In step 1, random coordinate vectors X_i are selected i.i.d. in $\mathbb{R}^d$ for each vertex i from a chosen distribution. In step 2, edges are created for each pair of vertices (i, j) with probability $X_i^T X_j$.

In [91], the authors suggest that the dot product models the idea of varying levels of talkativeness for social network members with the vertex vectors representing varying interests. Their work also extends RDPG for directed graphs.

With respect to the obtained graph properties, it has been shown that power law (in) degree distributions can be obtained by appropriately selecting the sampling distribution. Additionally, the diameter of the graph is largely constant for directed and undirected graphs and they show evidence of clustering. The two primary downsides to RDPG are its inability to create weighted graphs and a quadratic runtime (or O(n+m) for sparse graphs).

R-MAT [20] The R-MAT or **R**ecursive **MAT**rix model was first introduced by Chakrabati et al. [20]. Designed to be a simple, parsimonious model, it can quickly generate a diverse set of realistic graphs.

The basic algorithm operates on the adjacency matrix A where $a_{ij} = 1$ if there is an edge between i and j. Initially, all entries in A are 0. To create edges, the adjacency matrix is divided into 4 quadrants. The algorithm chooses a quadrant with probability a,b,c or d respectively where $a + b + c + d = 1$. This method recursively divides the selected quadrant (into 4 quadrants) until it reaches a 1x1 array where it "places" an edge. Duplicate edges may occur, thus the final number of edges may be less than anticipated.

R-MAT claims to be able to generate a wide range of graph types including directed, undirected, power-law, and bipartite graphs. The model can quickly generate large graphs, is readily available and easily parallelized. As such, it has been used in a wide variety of applications [38] and was selected for the Graph500 benchmark.

Small-World (Watts-Strogatz) [86] Vertices in small-world networks tend to have many "local" edges, and a relatively small numbers of "remote" edges. More formally, these graphs have a small average shortest path length, and a large clustering coefficient.

In [86], Watts and Strogatz introduced a straightforward method for generating graphs which exhibit this property. First, construct a ring with n vertices. Choose $k < n$, and connect each vertex to its k nearest neighbors. This is our starting point. Next, choose a rewiring probability, p. Make $k/2$ passes clockwise around the ring, visiting each vertex. On the i-th pass, re-wire, with probability p, the i^{th} edge closest to the current vertex to another randomly selected vertex.

TIERS [18, 27] Tiers is a network topology generator proposed by Matthew Doar as an improvement to the Waxman generator (and several variations thereof) in order to better reflect the hierarchical domain structure of the Internet. This is done using a three-level hierarchy which simulates Wide-Area (WAN), Metropolitan-Area (MAN) and Local-Area networks (LAN).

The total number of vertices N is a function of N_W (number of WAN), N_M (number of MAN), N_L (number of LAN), S_W (number of vertices per WAN), S_M (number of vertices per MAN), and S_L (number of vertices per LAN) where $N = N_W S_W + N_M S_M + N_M N_L S_L$. The model also allows for optional redundancy parameters and bandwidth.

At a high level, the algorithm first creates N_W transit domains with S_W vertices placed randomly in the grid. Edges are created using a minimum spanning tree then intra-network redundancy is corrected. The process is repeated for the MAN. To create the LAN networks, one vertex is chosen to be the center of the star and all others are connected to it. To connect these 3 levels together, each MAN is connected to the WAN with a single edge by randomly selecting vertices in the WAN. A similar procedure is followed connecting the star-center vertex in the LAN to the MAN. For additional algorithm details and caveats see the original paper [27]. One drawback of the current implementation is that it supports a maximum of 1 WAN.

Transit-stub [92] The transit-stub model described in [92] was an attempt to extend the state of the art in graph generation beyond the then-current practices of using purely random graphs, regular topologies, and existing real-world topologies. In particular, the authors looked at network topologies such as the Internet, and designed their model to approximate the hub-like properties they observed. In particular, transit-stub defines 2 different types of domains - *transit* and *stub*.

Stub domains can be thought of as networks where all traffic between any two vertices on a network stays within that network. *Transit* domains, on the other hand, connect stub domains to one another. This behavior reflects what is seen on the Internet - some domains (*stub* domains) are operated by a single entity, and traffic among that entity's nodes stay within the domain, whereas the *transit* domains represent interconnection points between different operators. See Figure 5 in [92] for an example of what this looks like in practice.

To construct a graph using the transit-stub model, generate a connected random graph, and then replace the vertices of that graph with another randomly generated graph. These new graphs are each a transit domain. For each vertex in these domains, we attach it to some number of new connected random graphs, and these become the stub domains connected to that transit domain. The final step is to create random edges between a transit domain and a stub, or between two stub domains.

Waxman [87] The Waxman generator is a geographic model where vertices are placed randomly and uniformly throughout a two-dimensional space and edges are created with probability p based on euclidean distance between vertices.

The Waxman generator was introduced as a means of generating synthetic graphs to use in analyzing network routing algorithms. As such, graphs generated in this manner should display characteristics that are more similar to real-world networks than generators such as Ërdos-Rényi .

Formally, the probability $P(\{u, v\})$ of creating an edge between vertices u and v is given by: $P(\{u, v\}) = \beta^{\frac{-d(u,v)}{L\alpha}}$ where $d(u, v)$ is simply the euclidean distances between vertices u and v. The parameter β controls the overall density of the graph and α dictates the ratio of short edges to long edges. L is the length of the longest edge in the graph.

A.2 OTHER MODELS

In this section we give a brief overview of other models we found, but did not test for various reasons.

ERGM [46] The Exponential Random Graph Model (ERGM) belongs to the exponential family of models. ERGMs create a way to study the probability of observing a set of relationships (edges) between a given set of actors (vertices). Formally, this is given by $P(Y = y) = \frac{exp\{\theta^T g(y)\}}{k(\theta)}$ where $g(y)$ is a vector of network statistics, θ is a vector of model parameters and $k(\theta)$ is a normalizing constant. Through appropriate model term choices things such as propensities for homophily, mutuality, and friend-of-a-friend triad closure can be simulated.

Typically, one starts with a set of network data and a (hypothesised) model (statistics of interest). From there, the maximum likelihood estimates (MLE) ($\hat{\theta}$) for the model are sought out. Direct computation of the MLEs is not feasible though, because $k(\theta)$ is difficult to compute, thus a Markov Chain Monte Carlo is employed. Given the outcome based on the supposed model and $\hat{\theta}$ parameters, the model fit can be assessed, the parameters interpreted and additional network realizations with the specified probabilities can be performed. It should be noted that these models can only represent (non) existence of edges (binary), thus limiting the scope. A large body of work exists for those

interested in more details on model creation, fitting and statistics of interest. We note however, that to generate synthetic graphs using this technique a pre-fitted and parameterized model is required.

PLRG [4] PLRG is a "curve fitting" power-law topology generator, which takes an explicit scale-free degree distribution, then interconnects the vertices to fit this "curve." Typically, the input sequence is generated as a set of independent random variables, each drawn from an underlying power law distribution (ignoring the correlation among degrees implied by their sum being required to equal twice the number of edges). Given an input sequence $(d_1, \ldots, d_n)$, PLRG first assigns the sequence to the n vertices of the graph, then randomly matches degrees among all the vertices. This may result in self-loops and duplicated links (and is also not guaranteed to give a connected graph as a result).

B DESCRIPTION OF PACKAGES

APGL [26] APGL is a general purpose graph library written in PYTHON. It provides a set of graph classes, kernels that operate on them, as well as additional classes to generate synthetic graphs. While APGL is no longer actively developed, we utilize it in order to provide additional implementations of models provided by other packages. Installation of APGL requires the NUMPY and SCIPY [49] libraries. Minimal modifications were made to the code, however we did add exception handling code to the edge removal functions in sparse graphs. This allowed us to use different matrix storage options that used less memory.

ergm [46] ERGM is part of the STATNET [42] statistical analysis package, and is written in R. It provides functions for working with exponential random graph models. ERGM installation requires a number of additional packages available from the CRAN, however no changes were required to the ERGM code.

GGEN [24] This package focuses on generating random directed acyclic graphs. Created in 2009, GGEN proposes to standardize the generation of random graphs for studying scheduling simulations in an effort to remove bias and errors when validating scheduling algorithms via simulation. Many of the classical methods are implemented including 2 versions of the Erdös-Rényi method ($G(n, p)$ and $G(n, M)$), layer-by-layer, fan-in/fan-out and random-orders.

Implemented in C++ on top of the BOOST graph library, the authors show that the generated graphs adhere to the desired statistics while remaining significantly different from each other.

GRAPH500 Written in C using OpenMP and MPI, this is the reference implementation of the Graph500 benchmark, used for generating RMAT and Kronecker graphs. The default output of the `make-edgelist` executable is a binary format; the code was modified to instead output an ASCII edgelist format.

gt-itm [93] GT-ITM is the Georgia Tech Internet Topology Models software package. It is written in C and primarily implements the transit-stub model. Installation requires the STANFORD GRAPH BASE [51] (SGB) library. Since GT-ITM is an older code, it required more changes than other generators. Fixes had to be made in the `Makefile` for it to compile correctly, as well as fixing a non-compliant linebreak in `edge.c`. Additionally, we created a new binary, `sgb2adjlist`, to convert the SGB output files to a more common and portable adjacency list format. We believe there may be bugs in GT-ITM or SGB that limit the maximum graph size. Finding and fixing these bugs is outside the scope of this paper.

INDDGO Written in C++, and using both OpenMP and MPI, INDDGO (Integrated Network Decomposition and Dynamic Programming for Graph Optimization) is an open source library for studying and leveraging the structure of large graphs. Originally designed to enable users to calculate and use tree decompositions for solving optimization problems (such as maximum weighted independent set), this framework has been significantly enhanced to enable the calculation and evaluation of a broad spectrum of graph features/statistics on both original networks and their associated tree decompositions (if desired). It supports output in Graphviz format to enable visualization of networks and vertex statistics, and integrates with third party libraries such as METIS, SuiteSparse, Boost, and SLEPc to improve efficiency. Utilities have also been written to enable easy conversion between different graph storage formats.

inet The INET package implements the inet model in C. Changes were made to use a random seed for the random number generator on every run, instead of the original default of 0. Functionality was also added to allow for basic edgelist output, as well as output to a file rather than `stdout`.

KRIOUKOV The KRIOUKOV generator is written in FORTRAN and is the only studied generator that is not publicly available. Changes were made to hard-coded limits in the source files to allow for larger graphs to be generated.

MFR A minimal free resolution (mfr) of a graph is a set of graph invariants. The MFR package, written in R, computes the mfr of the edge ideal of a graph and for some special graph types. We utilize functionality from the package to produce Random Dot Product Graphs.

MUSKETEER A Python code built on top of NETWORKX, MUSKETEER is a purpose-built code implementing the Musketeer model discussed previously. It may be necessary to convert the line endings in the files to be compatible with the operating system being used. The code is also modified to prevent output of a Python "pickle" object that is unused in our analysis.

NETWORKX [41] Like APGL, NETWORKX is a general purpose Python library for graph programming, and requires NUMPY and SCIPY. It provides an extensive array of graph analysis algorithms, as well as numerous input and output functions for retrieving and storing graphs to disk. The only major change made is the addition of a function to output edgelists in the format we desire.

pywebgraph PYWEBGRAPH, as the name implies, is written in Python. It generates a web-like graph using a threaded RMAT variant. It was necessary to modify the `setup.py` script to install all the necessary pre-requisite Python modules; otherwise, no changes are needed.

stochsim Written using Matlab, STOCHSIM is a stochastic simulation code. We utilize one of the examples to generate Waxman model graphs. Modifications were made to enable output in our preferred format.

tiers TIERS is a generator package written in an older dialect of C++. Due to this, we had to update some of the header includes, add a namespace, and change some variable declarations to `extern` in order to get it to compile correctly. Additionally, like many of the other generators, we added functions to output to a standardized edge list format.

Table 4. Generator packages

Package	Ver.	Obtained from	Lang.	Changes
APGL	v0.7.3	http://pythonhosted.org/apgl/	Python	provided as library: added wrapper script; required scipy and numpy
Boost	v.1.41	provided by OS	C++	no changes
BTER		http://www.sandia.gov/~tgkolda/bter_supplement/	Matlab	added wrapper script
ERGM	v.3.0-3	CRAN mirror (iastate)	R	added script to provide required model class as input and specific parameters as base
GGEN	v3.0.1	http://ligforge.imag.fr/frs/?group_id=77	C++	requires supporting libraries (boost)
Graph500	v2.1.4	http://graph500.org	C	added wrapper code
gt-itm		http://www.cc.gatech.edu/projects/gtitm/	C	requires SGB library; wrote Makefile; some code changes
INDDGO		https://github.com/bdsullivan/INDDGO	C++	none (for partial k-tree generation)
inet-3.0	v3.0	topology.eecs.umich.edu/inet/	C	none
Krioukov	n/a	provide by author	Fortran	increased size limits in code
MFR	v1.04	CRAN mirror (Indiana)	R	none
Musketeer	v1.02	http://people.cs.clemson.edu/~isafro/musketeer/index.html	Python	added timing information
NetworkX	v1.7	http://networkx.lanl.gov/	Python	provided as library: added wrapper script
pywebgraph	v2.72	http://pywebgraph.sourceforge.net/	Python	changes to install modules correctly
stocksim	v2.1	http://www2.math.uu.se/research/telecom/software/	Matlab	removed plotting; added calculation of matrix memory usage; added wrapper to store results to file
Tiers	v1.1	http://www.isi.edu/nsnam/ns/ns-topogen.html#tiers	C++	replaced deprecated header files; defined variables in source file to handle "extern" variables

7. References

[1] Brief introduction | graph 500, 2012. http://www.graph500.org/.

[2] Introduction and objectives | TOP500 supercomputer sites, 2013. http://www.top500.org/project/introduction/.

[3] PyPy's speed center, 2013. http://speed.pypy.org/.

[4] Aiello, W., Chung, F., and Lu, L. A random graph model for power law graphs. *Experimental Mathematics 10*, 1 (2001), 53–66.

[5] Albert, R., and Barabasi, A.-L. Statistical mechanics of complex networks. *arXiv:cond-mat/0106096* (June 2001). Reviews of Modern Physics 74, 47 (2002).

[6] Alon, N., and Gutner, S. Linear time algorithms for finding a dominating set of fixed size in degenerated graphs. *Algorithmica 54*, 4 (Aug. 2009), 544–556.

[7] Alon, N., Yuster, R., and Zwick, U. Finding and counting given length cycles. *Algorithmica 17* (1997), 354–364.

[8] Anantharaj, V., Foertter, F., Joubert, W., and Wells, J. APPROACHING EXASCALE: application requirements for OLCF leadership computing. Tech. Rep. ORNL/TM-2013/186, July 2013.

[9] Babai, L., Grigoryev, D. Y., and Mount, D. M. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1982), STOC '82, ACM, p. 310–324.

[10] Bader, D. Graph 500 benchmark 1 ("Search"), 2012. http://www.cc.gatech.edu/~jriedy/tmp/graph500/.

[11] Bader, D. A., and Madduri, K. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *The 12th International Conference on High Performance Computing (HiPC 2005)* (2005), Springer, p. 465–476.

[12] Bader, D. A., and Madduri, K. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Computing 34*, 11 (Nov. 2008), 627–639.

[13] Barabasi, A.-L., and Albert, R. Emergence of scaling in random networks. *arXiv:cond-mat/9910332* (Oct. 1999). Science 286, 509 (1999).

[14] Batagelj, V., and Brandes, U. Efficient generation of large random networks. *Physical Review E 71*, 3 (Mar. 2005), 036113.

[15] Batagelj, V., and Zaversnik, M. An o(m) algorithm for cores decomposition of networks. arXiv e-print cs/0310049, Oct. 2003. Advances in Data Analysis and Classification, 2011. Volume 5, Number 2, 129-145.

[16] Beamer, S., Buluc, A., Asanovic, K., and Patterson, D. Distributed memory breadth-first search revisited: Enabling bottom-up search. Tech. Rep. UCB/EECS-2013-2, EECS Department, University of California, Berkeley, Jan. 2013.

[17] Brandes, U. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology 25*, 2 (2001), 163–177.

[18] Calvert, K., Doar, M., and Zegura, E. Modeling internet topology. *IEEE Communications Magazine 35*, 6 (1997), 160–163.

[19] Chakrabarti, D., and Faloutsos, C. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv. 38*, 1 (June 2006).

[20] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: a recursive model for graph mining. In *In SDM* (2004).

[21] Clauset, A., Shalizi, C. R., and Newman, M. E. J. Power-law distributions in empirical data. arXiv e-print 0706.1062, June 2007. SIAM Review 51, 661-703 (2009).

[22] Cohen, N., Coudert, D., and Lancin, A. Exact and approximate algorithms for computing the hyperbolicity of large-scale graphs.

[23] Colella, P. Defining software requirements for scientific computing, 2004. http://view.eecs.berkeley.edu/w/images/temp/6/6e/20061003235551!DARPAHPCS.ppt.

[24] Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.-M., and Wagner, F. Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques* (2010), p. 60.

[25] Courcelle, B. Handbook of theoretical computer science (vol. b). MIT Press, Cambridge, MA, USA, 1990, p. 193–242.

[26] Dhanjal, C. An introduction to APGL. *Statistics and Applications Group, Telecom ParisTech* (2010).

[27] Doar, M. A better model for generating test networks. In *Global Telecommunications Conference, 1996. GLOBECOM '96. 'Communications: The Key to Global Prosperity* (1996), pp. 86–93.

[28] Dongarra, J. J., and Luszczek, P. R. HPCC, 2013. http://icl.cs.utk.edu/hpcc/.

[29] Downey, R. G., and Fellows, M. R. *Parameterized complexity [...] [...]*. Springer, New York [u.a., 1999.

[30] Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., and Woodhull, G. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE* (2003), Springer-Verlag, p. 127–148.

[31] Erdős, P., and Rényi, A. On random graphs i. *Publicationes Mathematicae 6*, Publicationes Mathematicae 6 (1959), 290–297.

[32] Erdős, P., and Rényi, A. On the evolution of random graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES* (1960), p. 17–61.

[33] Faloutsos, M., Faloutsos, P., and Faloutsos, C. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev. 29*, 4 (Aug. 1999), 251–262.

[34] Feng, W.-C., and Cameron, K. About | the green500. http://www.green500.org/about.

[35] Fournier, H., Ismail, A., and Vigneron, A. Computing the gromov hyperbolicity of a discrete metric space. arXiv e-print 1210.3323, Oct. 2012.

[36] Garey, M. R., and Johnson, D. S. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman, San Francisco, 1979.

[37] Gilbert, E. N. Random graphs. *The Annals of Mathematical Statistics 30*, 4 (Dec. 1959), 1141–1144.

[38] Groër, C., Sullivan, B. D., and Poole, S. A mathematical analysis of the r-MAT random graph generator. *Networks 58*, 3 (2011), 159–170.

[39] Groer, C. S., Sullivan, B. D., and Weerapurage, D. P. INDDGO: integrated network decomposition & dynamic programming for graph optimization. Tech. rep., Oct. 2012.

[40] Gutfraind, A., Meyers, L. A., and Safro, I. Multiscale network generation. *arXiv:1207.4266* (July 2012).

[41] Hagberg, A., Schult, D., and Swart, P. Exploring network structure, dynamics, and function using NetworkX. G. Varoquaux, T. Vaught, and J. Millman, Eds., pp. 11–15.

[42] Handcock, M. S., Hunter, D. R., Butts, C. T., Goodreau, S. M., and Morris, M. statnet: Software tools for the statistical modeling of network data, 2003. `http://statnetproject.org`.

[43] Hernandez, J. M., Kleigerg, T., Wang, H., and Van Mieghem, P. A comparison of topology generators with power law behavior. In *Proceedings of SPECTS* (2007), pp. 484–493.

[44] Hernandez, V., Roman, J. E., and Vidal, V. SLEPc: a scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Softw. 31*, 3 (Sept. 2005), 351–362.

[45] Heroux, M. A., and Dongarra, J. Toward a new metric for ranking high performance computing systems. *SAND2013 - 4744* (June 2013).

[46] Hunter, D. R., Handcock, M. S., Butts, C. T., Goodreau, S. M., and Morris, M. ergm: A package to fit, simulate and diagnose exponential-family models for networks. *Journal of Statistical Software* (2008).

[47] Jin, C., Chen, Q., and Jamin, S. *Inet: Internet Topology Generator*. 2000.

[48] Johnson, G. HPCwire: the HPC triple crown, Nov. 2012. `http://www.hpcwire.com/hpcwire/2012-11-28/the_hpc_triple_crown.html?page=1&featured=top&adclass=FrontPage`.

[49] Jones, E., Oliphant, T., Peterson, P., et al. *SciPy: Open source scientific tools for Python*. 2001. = http://www.scipy.org/.

[50] Kloks, T. *Treewidth: Computations and Approximations*, vol. 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[51] Knuth, D. E. *The Stanford GraphBase: a platform for combinatorial computing*. ACM, New York, NY, USA, 1993.

[52] Kolda, T. G., Pinar, A., Plantenga, T., and Seshadhri, C. A scalable generative graph model with community structure. *arXiv:1302.6636* (Feb. 2013).

[53] Kramer, W. Top problems with the TOP500, Nov. 2012. http://www.ncsa.illinois.edu/News/Stories/TOP500problem/.

[54] Krioukov, D., Papadopoulos, F., Kitsak, M., Vahdat, A., and Boguñá, M. Hyperbolic geometry of complex networks. *Physical Review E 82*, 3 (Sept. 2010), 036106.

[55] Latapy, M. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science 407*, 1–3 (Nov. 2008), 458–473.

[56] Leskovec, J., Chakrabarti, D., Kleinberg, J., and Faloutsos, C. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *Knowledge Discovery in Databases: PKDD 2005*, A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, Eds., no. 3721 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2005, pp. 133–145.

[57] Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., and Ghahramani, Z. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res. 11* (Mar. 2010), 985–1042.

[58] Leskovec, J., and Faloutsos, C. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 631–636.

[59] Leskovec, J., and Faloutsos, C. Scalable modeling of real graphs using kronecker multiplication. In *Proceedings of the 24th international conference on Machine learning* (New York, NY, USA, 2007), ICML '07, ACM, p. 497–504.

[60] Leskovec, J., Kleinberg, J., and Faloutsos, C. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005), p. 177–187.

[61] Lick, D. R., and White, A. T. k-degenerate graphs. *Canadian Journal of Mathematics 22*, 0 (Jan. 1970), 1082–1096.

[62] Lothian, J., Kuehn, J., Baker, M., Schrock, J., and Poole, S. Systemburn technical report. Tech. Rep. ORNL/TM-2013/234, June 2013.

[63] Luks, E. M. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1980), IEEE Computer Society, p. 42–49.

[64] Luszczek, P., Dongarra, J. J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., Mccalpin, J., Bailey, D., and Takahashi, D. Introduction to the HPC challenge benchmark suite. Tech. rep., 2005.

[65] Mahadevan, P., Krioukov, D., Fall, K., and Vahdat, A. Systematic topology analysis and generation using degree correlations. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2006), SIGCOMM '06, ACM, pp. 135–146.

[66] Miller, J. C., and Hagberg, A. Efficient generation of networks with given expected degrees. In *Proceedings of the 8th international conference on Algorithms and models for the web graph* (Berlin, Heidelberg, 2011), WAW'11, Springer-Verlag, pp. 115–126.

[67] Mishra, R., Shukla, S., Arora, D. D., and Kumar, M. An effective comparison of graph clustering algorithms via random graphs. *International Journal of Computer Applications 22*, 1 (May 2011), 22–27. Published by Foundation of Computer Science.

[68] NCI. NCI national facility - top500, 2013. `http://nf.nci.org.au/notices_news/top500.php`.

[69] Newman, M. E. J. Mixing patterns in networks. *Physical Review E 67*, 2 (Feb. 2003), 026126.

[70] Newman, M. E. J. The structure and function of complex networks. *SIAM REVIEW 45* (2003), 167–256.

[71] Pennock, D. M., Flake, G. W., Lawrence, S., Glover, E. J., and Giles, C. L. Winners don't take all: Characterizing the competition for links on the web. *Proceedings of the national academy of sciences 99*, 8 (2002), 5207–5211.

[72] Philip, G., Raman, V., and Sikdar, S. Polynomial kernels for dominating set in graphs of bounded degeneracy and beyond. *ACM Trans. Algorithms 9*, 1 (Dec. 2012), 11:1–11:23.

[73] Pinto, D. M., and Markenzon, L. New concepts and results on the average degree of a graph. *Applicable Analysis and Discrete Mathematics 1*, 1 (2007), 284–292.

[74] Potamias, M., Bonchi, F., Castillo, C., and Gionis, A. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management* (2009), p. 867–876.

[75] Raghavendra, P., and Steurer, D. Graph expansion and the unique games conjecture. In *Proceedings of the 42nd ACM symposium on Theory of computing* (2010), p. 755–764.

[76] Scheinerman, E. R., and Tucker, K. Modeling graphs using dot product representations. *Computational Statistics 25*, 1 (Mar. 2010), 1–16.

[77] Seshadhri, C., Kolda, T. G., and Pinar, A. Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E 85*, 5 (May 2012), 056109.

[78] Seshadhri, C., Pinar, A., and Kolda, T. G. An in-depth analysis of stochastic kronecker graphs. *arXiv:1102.5046* (Feb. 2011).

[79] Shah, J. V., Poon, C.-S., Wang, J., and Malakooti, B. Observations on characterization of training errors in supervised learning using gradient-based rules, and authors' response. *Neural Netw. 8*, 4 (June 1995), 659–660.

[80] Simon, H. A. On a class of skew distribution functions. *Biometrika 42*, 3-4 (Dec. 1955), 425–440.

[81] Spencer, J. Nine lectures on random graphs. In *Ecole d'Eté de Probabilités de Saint-Flour XXI - 1991*, P.-L. Hennequin, Ed., no. 1541 in Lecture Notes in Mathematics. Springer Berlin Heidelberg, Jan. 1993, pp. 293–347.

[82] Subramaniam, B., and Feng, W.-c. The green index: A metric for evaluating system-wide energy efficiency in HPC systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International* (2012), pp. 1007–1013.

[83] Takes, F., and Kosters, W. Computing the eccentricity distribution of large graphs. *Algorithms 6*, 1 (Feb. 2013), 100–118.

[84] Tangmunarunkit, H., Govindan, R., Jamin, S., Shenker, S., and Willinger, W. Network topology generators: degree-based vs. structural. *SIGCOMM Comput. Commun. Rev. 32*, 4 (Aug. 2002), 147–159.

[85] Ueno, K., and Suzumura, T. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2012), HPDC '12, ACM, p. 149–160.

[86] Watts, D. J., and Strogatz, S. H. Collective dynamics of 'small-world' networks. *Nature 393*, 6684 (June 1998), 440–442.

[87] Waxman, B. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications 6*, 9 (Dec. 1988), 1617–1622.

[88] Winick, J., and Jamin, S. Inet-3.0: Internet topology generator. Tech. rep., Technical Report CSE-TR-456-02, University of Michigan, 2002.

[89] Wormald, N. C. Models of random regular graphs. *London Mathematical Society Lecture Note Series* (1999), 239–298.

[90] Ye, Q., Wu, B., and Wang, B. Distance distribution and average shortest path length estimation in real-world networks. In *Advanced Data Mining and Applications*, L. Cao, Y. Feng, and J. Zhong, Eds., no. 6440 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2010, pp. 322–333.

[91] Young, S. J., and Scheinerman, E. R. Random dot product graph models for social network. In *OF LECTURE NOTES IN COMPUTER SCIENCE* (2007), pp. 138–149.

[92] Zegura, E., Calvert, K., and Bhattacharjee, S. How to model an internetwork. In *Proceedings IEEE INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation* (Mar. 1996), vol. 2, pp. 594–602 vol.2.

[93] Zegura, E. W. Modeling topology of internetworks home page. `http://www.cc.gatech.edu/projects/gtitm/`.

[94] Zwick, U. Exact and approximate distances in graphs—a survey. In *Algorithms—ESA 2001*. Springer, 2001, p. 33–48.