Scalable Generation of Graphs for Benchmarking HPC Community-Detection Algorithms

George M. Slota[1] Jonathan Berry[2]
Simon D. Hammond[2] Stephen L. Olivier[2]
Cynthia Phillips[2] Siva Rajamanickam[2]

[1]Rensselaer Polytechnic Institute, [2]Sandia National Labs

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Highlights: Adapted BTER

Primary results of this work:

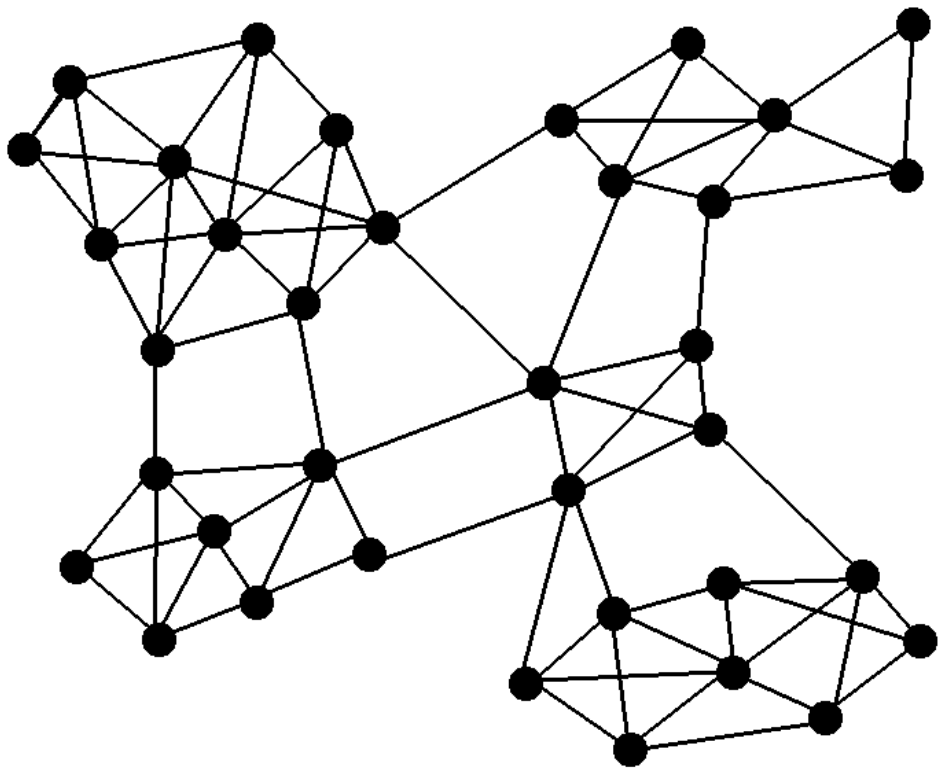
- Novel: First scalable approach for HPC community detection benchmarking via "engineered solutions"
  – We also develop a novel approach for *scaling* degree and clustering coefficient distributions
- Realistic: We utilize *real-world* graph distributions with a BTER implementation[1] and *edge-skipping*[2]
- Fast: 1 trillion edges/minute on current supercomputers
  – **Orders-of-magnitude faster than state-of-the-art**

We call our approach "Adapted BTER"

[1] [Kolda et al., 2014]
[2] [Miller and Hagberg, 2011]

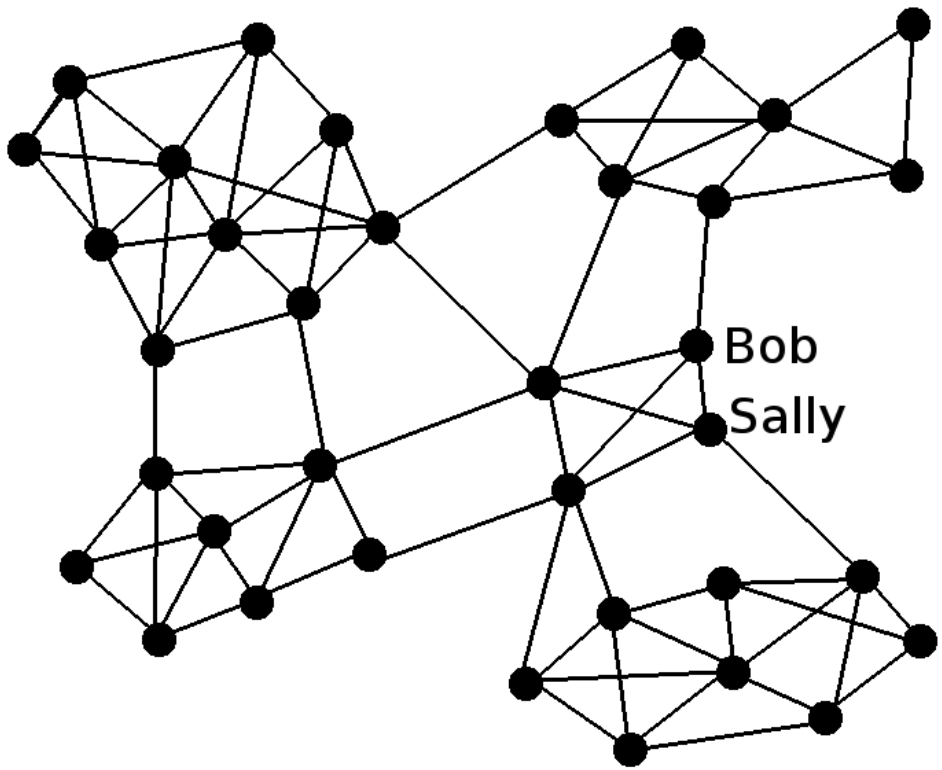
What is community detection?
Community Detection: Basic problem

- We have some real-world interaction network (e.g., Facebook)



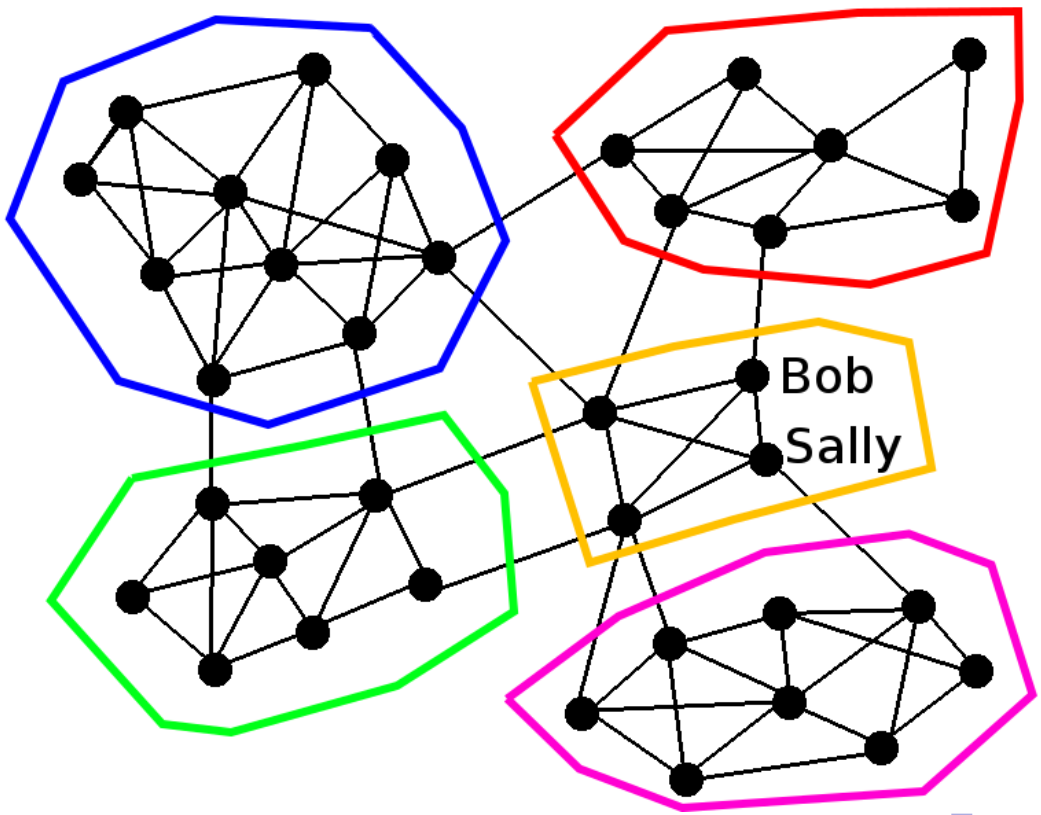
What is community detection?
Community Detection: Basic problem

- We have some real-world interaction network (e.g., Facebook)



What is community detection?
Community Detection: Basic problem

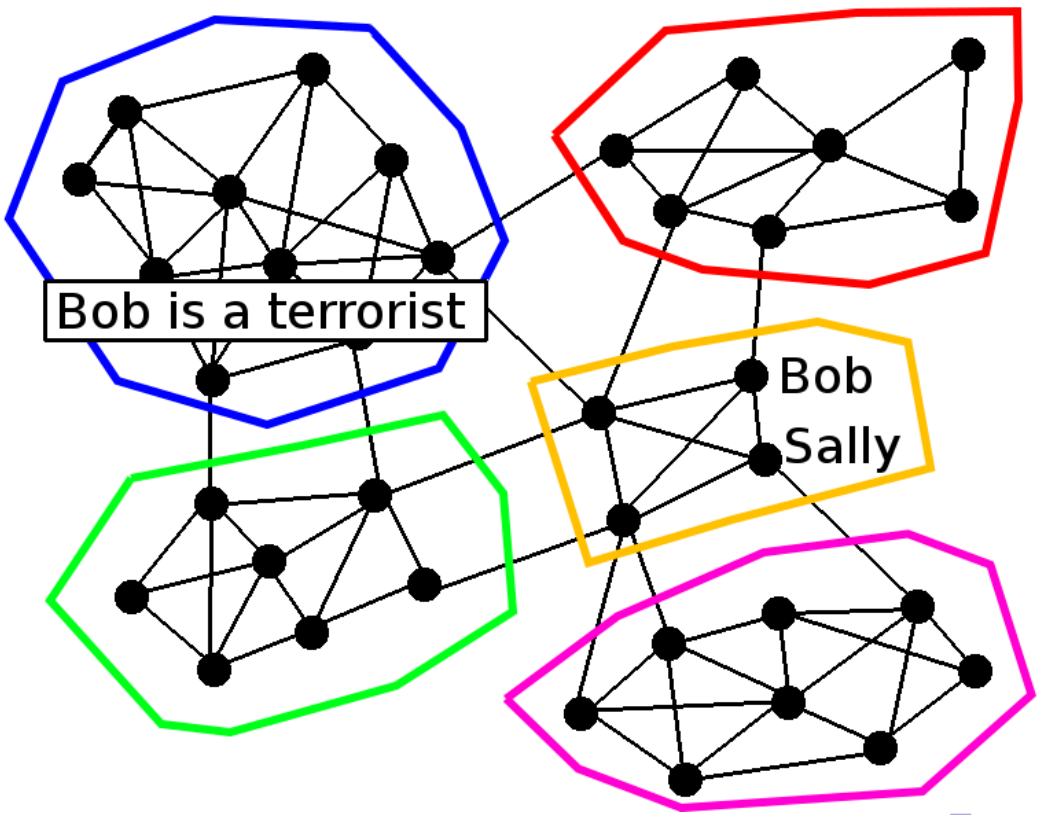
- We have some real-world interaction network (e.g., Facebook)
- Community detection: identifying *clusters* within the network



What is community detection?

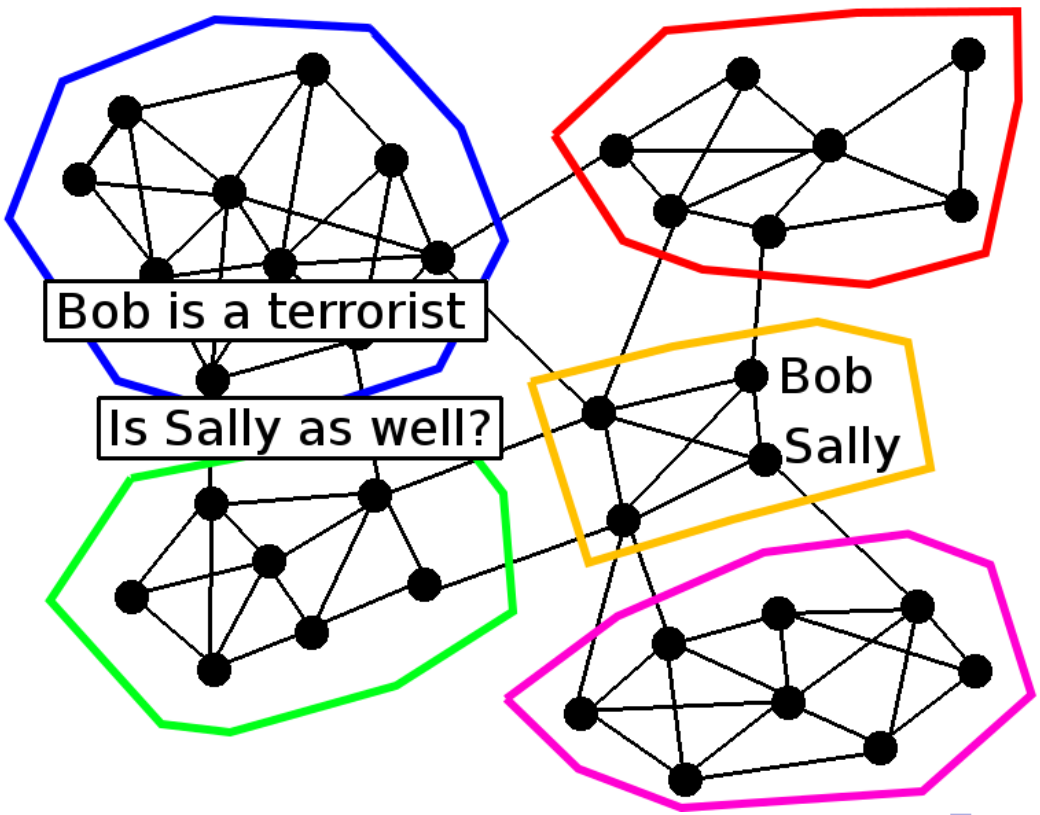
Community Detection: Basic problem

- We have some real-world interaction network (e.g., Facebook)
- Community detection: identifying *clusters* within the network
- Why: communities are often homogeneous (like-attracts-like) so we can often infer information about community members.



What is community detection?
Community Detection: Basic problem

- We have some real-world interaction network (e.g., Facebook)
- Community detection: identifying *clusters* within the network
- Why: communities are often homogeneous (like-attracts-like) so we can often infer information about community members.



How do we evaluate algorithm solution quality?

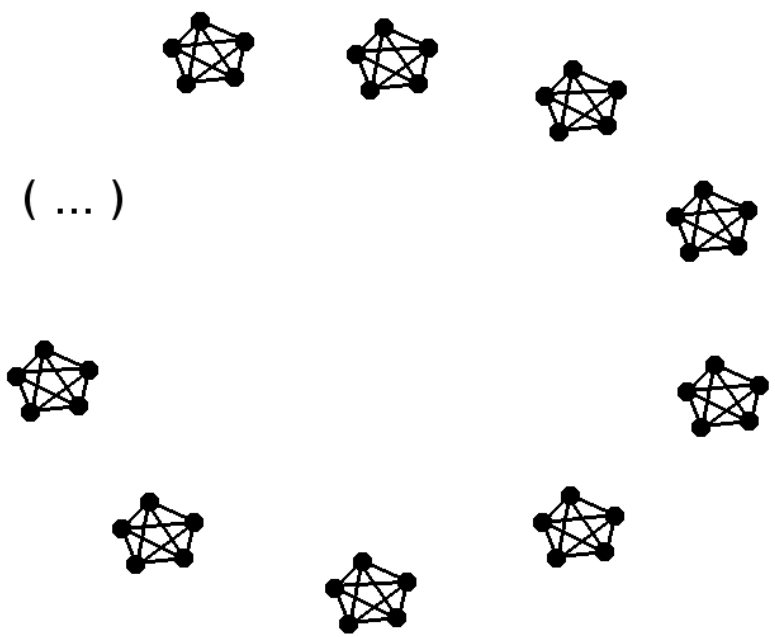
Community Detection Algorithms: Evaluation

Given some community detection algorithm, how can we determine the quality of its output?

- **Ideally:** Evaluate on real-world datasets with "known" communities
  – Very few such datasets exists, none at HPC/real-world social network scale
- **Small scale:** Generate synthetic networks with an "approximate engineered solution" (EAS) as communities
  – Until this current work, infeasible to generate and evaluate at a large scale
- **Large scale:** Calculate some global measurement such as modularity (how well-clustered is the solution versus random expectation)
  – For modularity in particular, *this approach is rather flawed*

The problem with evaluating with modularity

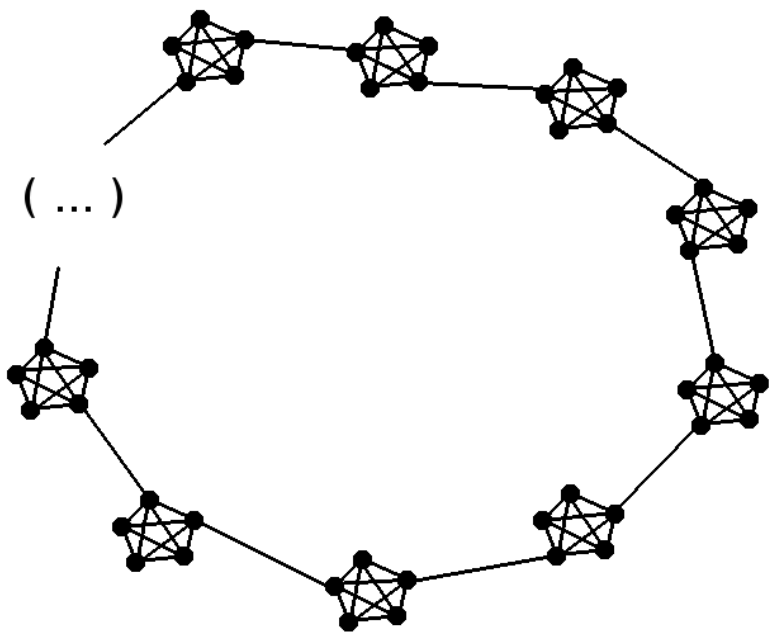
- Modularity suffers from a "resolution limit"
  - Small well-defined communities can not be individually resolved
  - Example: maximizing modularity on ring of cliques
    – cliques converge into single communities against intuition



The problem with evaluating with modularity

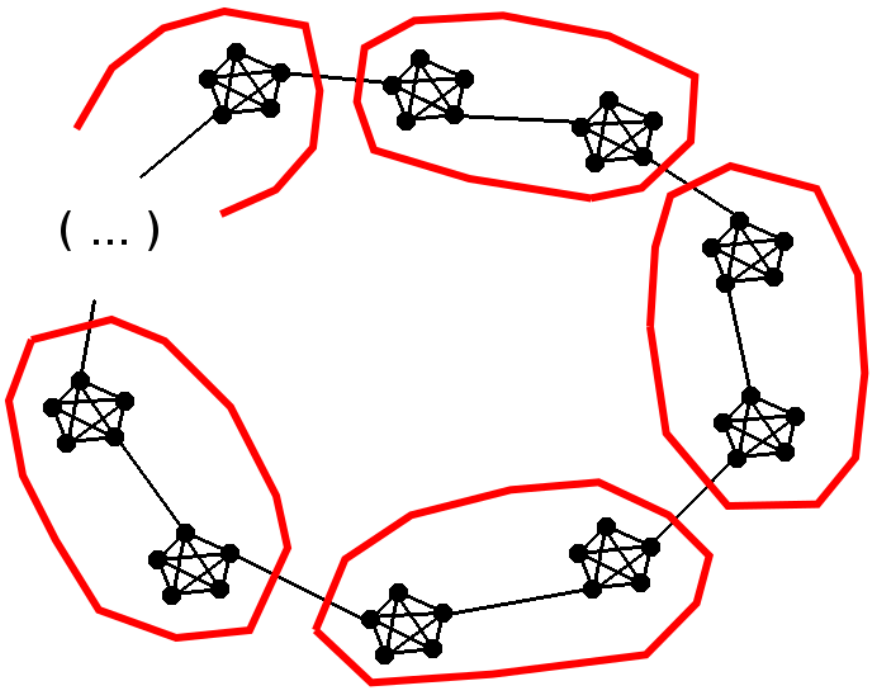
Community Detection Algorithms: Evaluating with Modularity

- Modularity suffers from a "resolution limit"
  - Small well-defined communities can not be individually resolved
  - Example: maximizing modularity on ring of cliques
    – cliques converge into single communities against intuition



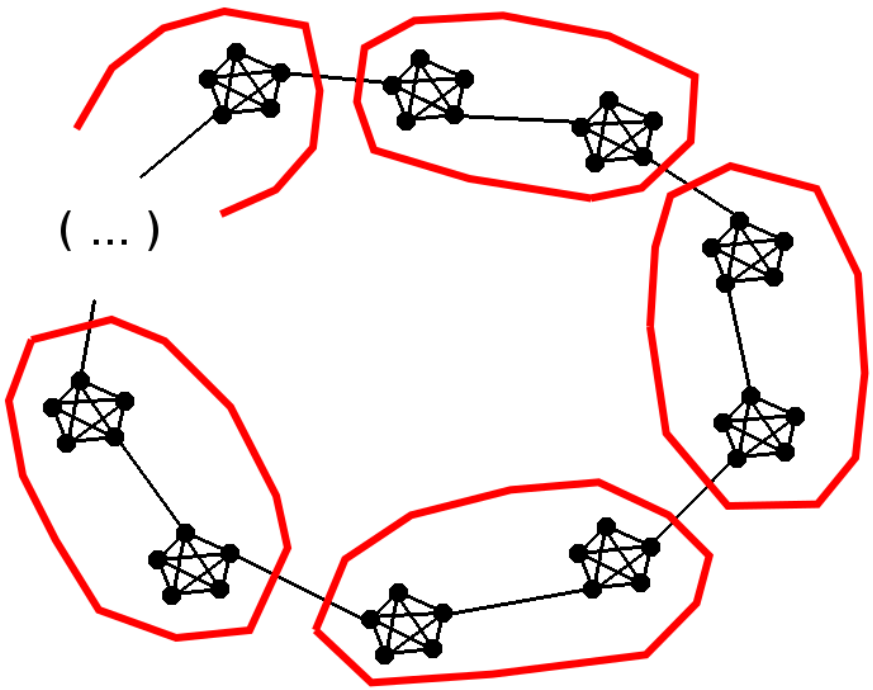
The problem with evaluating with modularity

- Modularity suffers from a "resolution limit"
  - Small well-defined communities can not be individually resolved
  - Example: maximizing modularity on ring of cliques
    – cliques converge into single communities against intuition



The problem with evaluating with modularity

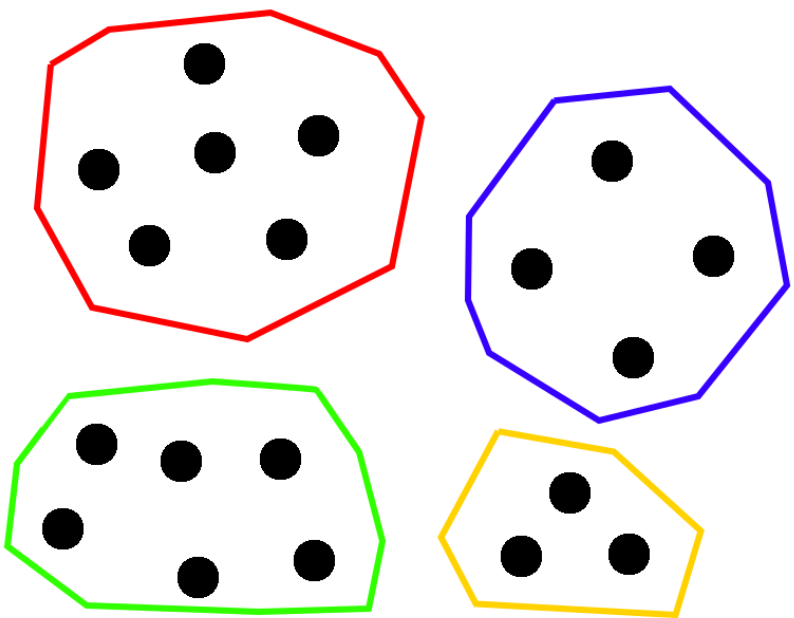
- Modularity suffers from a "resolution limit"
  - Small well-defined communities can not be individually resolved
  - Example: maximizing modularity on ring of cliques
    – cliques converge into single communities against intuition
- Real-world networks scale to billions of vertices
  – Yet human "community" sizes tend to be relatively constant
- Takeaway: **higher modularity != higher solution quality**



Using an "engineered approximate solution" (EAS)

Community Detection Algorithms: Evaluating with EAS instead

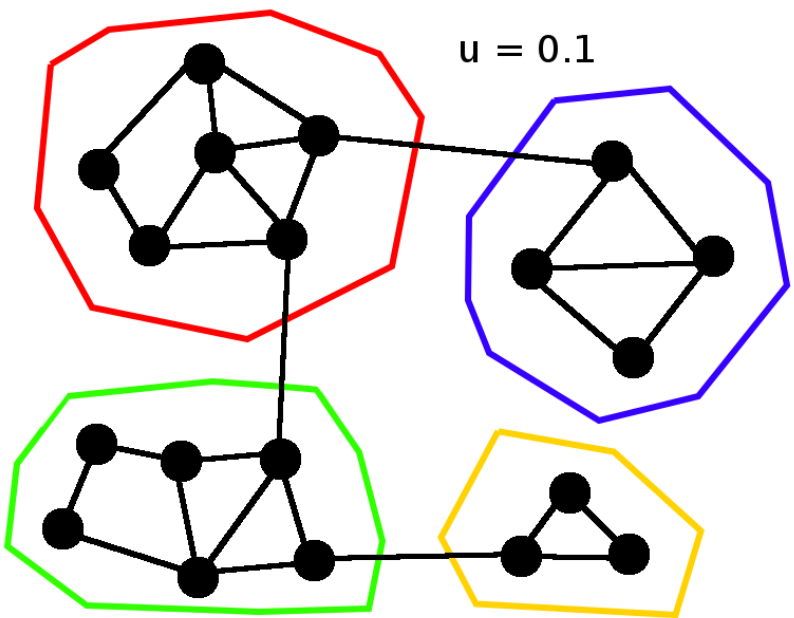
- Generate a synthetic network with some set of engineered "communities"



Using an "engineered approximate solution" (EAS)

Community Detection Algorithms: Evaluating with EAS instead

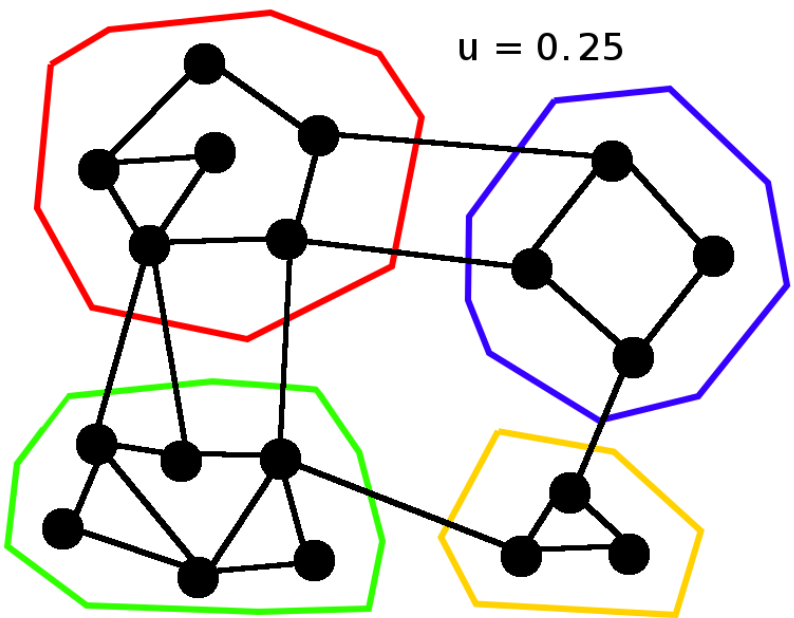
- Generate a synthetic network with some set of engineered "communities"
- Include a *mixing parameter* – μ – that controls the ratio of inter- to intra-community edges: $\mu \approx \frac{\text{inter-comm. edges}}{\text{total edges}}$
  – Effectively, this determines how well-defined the communities are



Using an "engineered approximate solution" (EAS)

Community Detection Algorithms: Evaluating with EAS instead

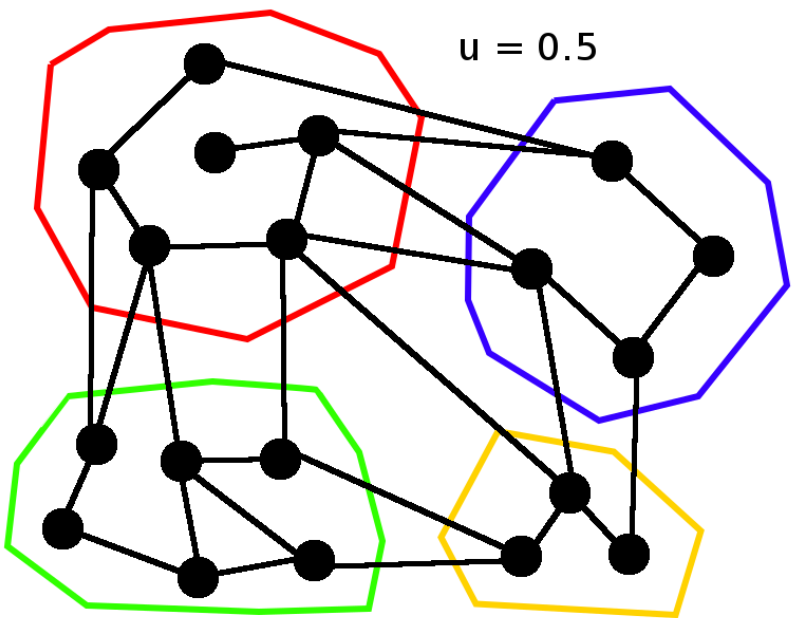
- Generate a synthetic network with some set of engineered "communities"
- Include a *mixing parameter* – μ – that controls the ratio of inter- to intra-community edges: $\mu \approx \frac{\text{inter-comm. edges}}{\text{total edges}}$
  – Effectively, this determines how well-defined the communities are



Using an "engineered approximate solution" (EAS)

Community Detection Algorithms: Evaluating with EAS instead

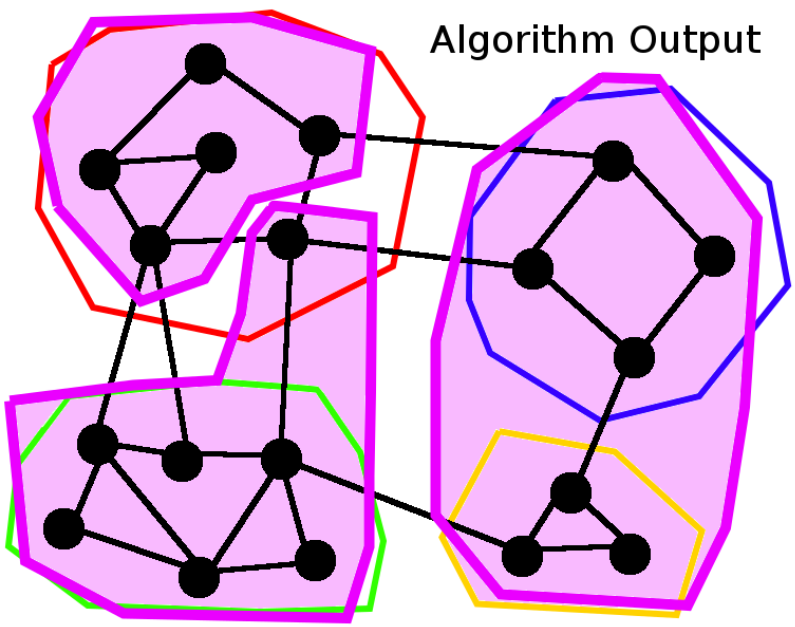
- Generate a synthetic network with some set of engineered "communities"
- Include a *mixing parameter* – μ – that controls the ratio of inter- to intra-community edges: $\mu \approx \frac{\text{inter-comm. edges}}{\text{total edges}}$
  – Effectively, this determines how well-defined the communities are



Using an "engineered approximate solution" (EAS)

Community Detection Algorithms: Evaluating with EAS instead

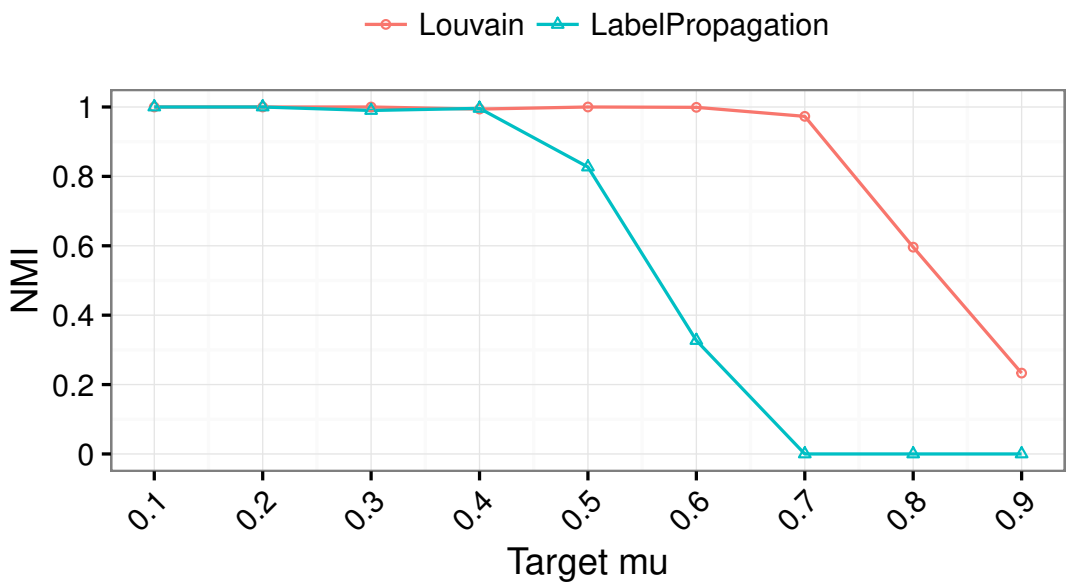
- Generate a synthetic network with some set of engineered "communities"
- Include a *mixing parameter* – μ – that controls the ratio of inter- to intra-community edges: $\mu \approx \frac{\text{inter-comm. edges}}{\text{total edges}}$
  – Effectively, this determines how well-defined the communities are
- Evaluate how well an algorithm's output matches the defined solution
  – Commonly utilize Normalized Mutual Information (NMI)



Using an "engineered approximate solution" (EAS)

Community Detection Algorithms: Evaluating with EAS instead

- Generate a synthetic network with some set of engineered "communities"
- Include a *mixing parameter* – μ – that controls the ratio of inter- to intra-community edges: $\mu \approx \frac{\text{inter-comm. edges}}{\text{total edges}}$
  – Effectively, this determines how well-defined the communities are
- Evaluate how well an algorithm's output matches the defined solution
  – Commonly utilize Normalized Mutual Information (NMI)
- Compare how well algorithms perform as you increase edge mixing via μ



Current State-of-the-Art: LFR

For benchmark graph generation with engineered solutions

"Lancichinetti–Fortunato–Radicchi" (LFR)[3]:

- With $>$1600 citations, this is a de facto standard
- Generates approximate solution to test against
  – **Uses tunable parameter for community coherence:** μ
- Limited scalability: best implementation takes ~17hrs to generate ~10B edges[4]
  – Original code takes hours for million+ edge graphs

Our Goal: *Develop methods to evaluate algorithms at HPC scale against an "engineered approximate solution."*

[3] [Lancichinetti et al., 2008]
[4] [Hamann et al., 2018]

A-BTER: Adapted BTER

Our full approach for HPC-scale benchmark generation and evaluation

Input: Real or synthetic degree and clustering coefficient distributions

1. Optional step: Scale the input degree and clustering coefficient distributions
2. Solve linear program to (further) shift clustering coefficient distributions to match some target μ_g
3. Pass new distributions to an efficient edge-skipping based BTER implementation
4. Run community detection algorithm on generated graph, evaluate versus "engineered approximate solution" (EAS)

Output: A measure of algorithm solution quality

A-BTER: Adapted BTER

Our full approach for HPC-scale benchmark generation and evaluation

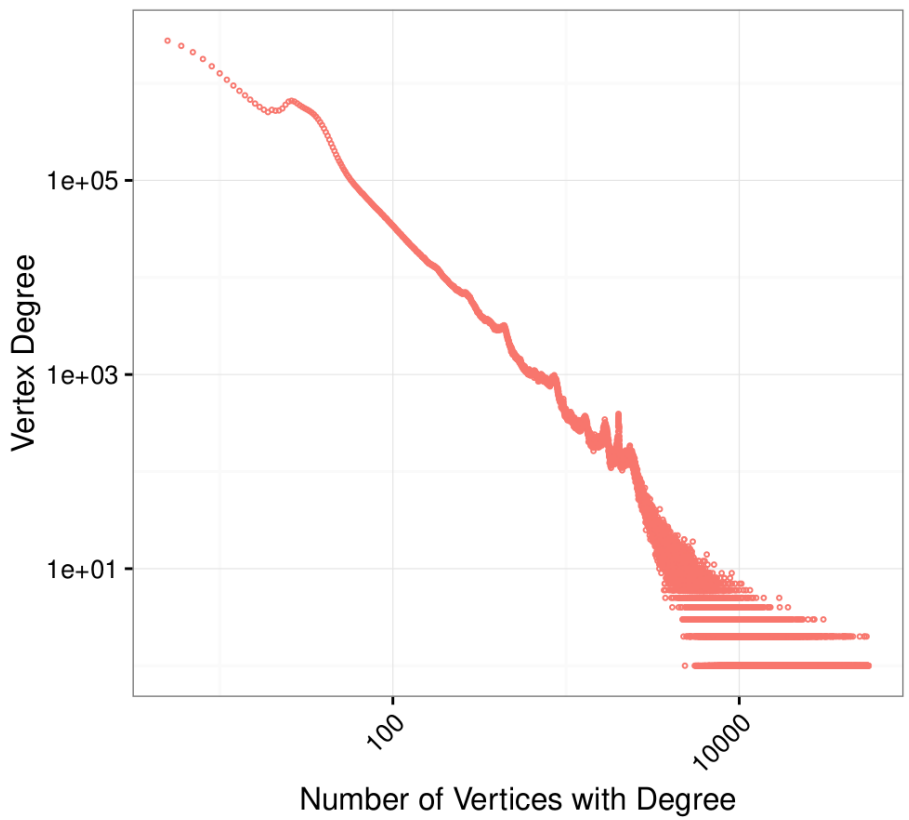
Input: Real or synthetic degree and clustering coefficient distributions

1. **Optional step: Scale the input degree and clustering coefficient distributions**
2. Solve linear program to (further) shift clustering coefficient distribution to match some target μ_g
3. Pass new distributions to an efficient edge-skipping based BTER implementation
4. Run community detection algorithm on generated graph, evaluate versus "engineered approximate solution" (EAS)

Output: A measure of algorithm solution quality

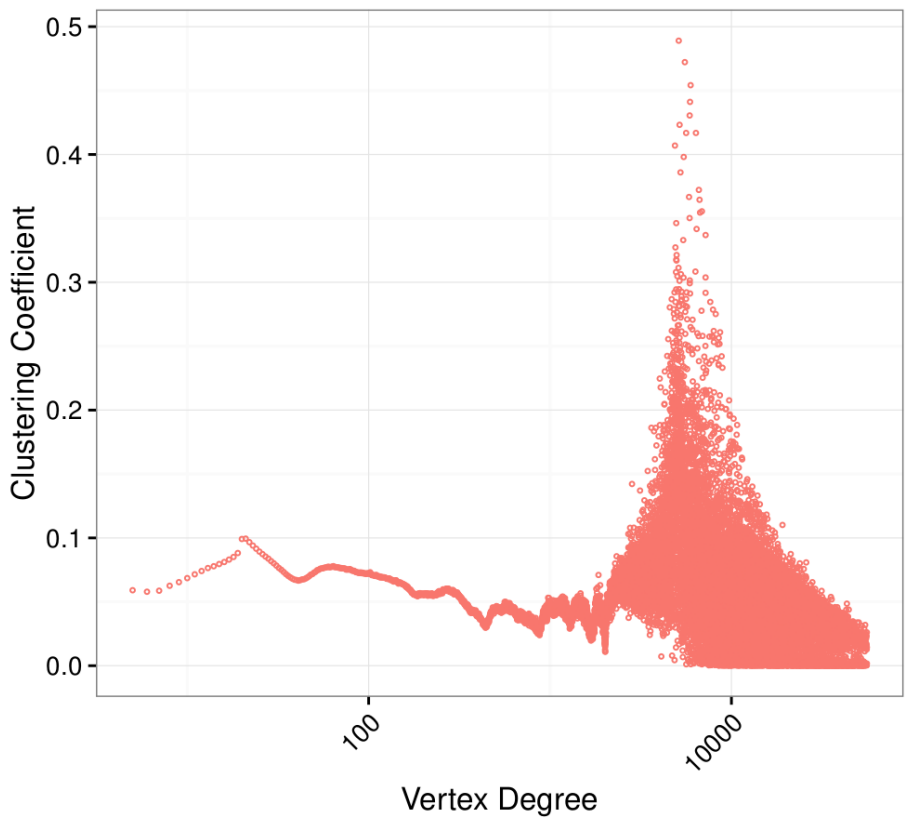
Background:
Degree and Clustering Coefficient Distributions

- **Degree Distribution:** How many vertices in the graph with each degree?



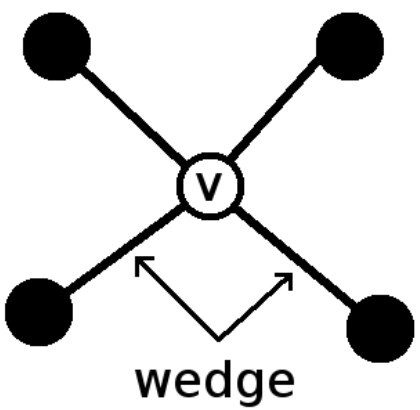
Background:
Degree and Clustering Coefficient Distributions

- **Degree Distribution:** How many vertices in the graph with each degree?
- **Clustering Coefficient Distribution:** What is the average clustering coefficient for each unique degree?



Background:
Degree and Clustering Coefficient Distributions

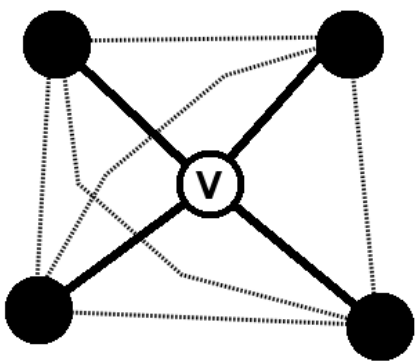
- **Degree Distribution:** How many vertices in the graph with each degree?
- **Clustering Coefficient Distribution:** What is the average clustering coefficient for each unique degree?
- **Clustering Coefficient:** Fraction of my friends that are friends with each other



$$CC = \frac{\text{closed wedges}}{\text{all wedges}}$$

Background:
Degree and Clustering Coefficient Distributions

- **Degree Distribution:** How many vertices in the graph with each degree?
- **Clustering Coefficient Distribution:** What is the average clustering coefficient for each unique degree?
- **Clustering Coefficient:** Fraction of my friends that are friends with each other

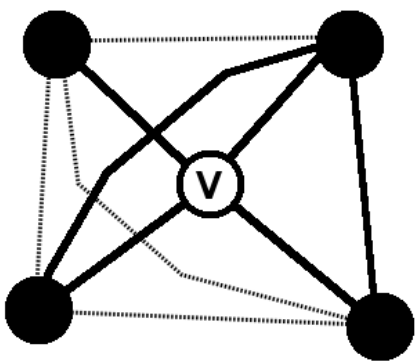


$$CC = \frac{\text{closed wedges}}{\text{all wedges}}$$

$$CC_v = \frac{0}{6}$$

Background:
Degree and Clustering Coefficient Distributions

- **Degree Distribution:** How many vertices in the graph with each degree?
- **Clustering Coefficient Distribution:** What is the average clustering coefficient for each unique degree?
- **Clustering Coefficient:** Fraction of my friends that are friends with each other

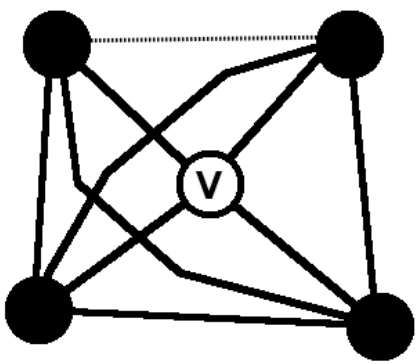


$$CC = \frac{\text{closed wedges}}{\text{all wedges}}$$

$$CC_v = \frac{2}{6}$$

Background:
Degree and Clustering Coefficient Distributions

- **Degree Distribution:** How many vertices in the graph with each degree?
- **Clustering Coefficient Distribution:** What is the average clustering coefficient for each unique degree?
- **Clustering Coefficient:** Fraction of my friends that are friends with each other



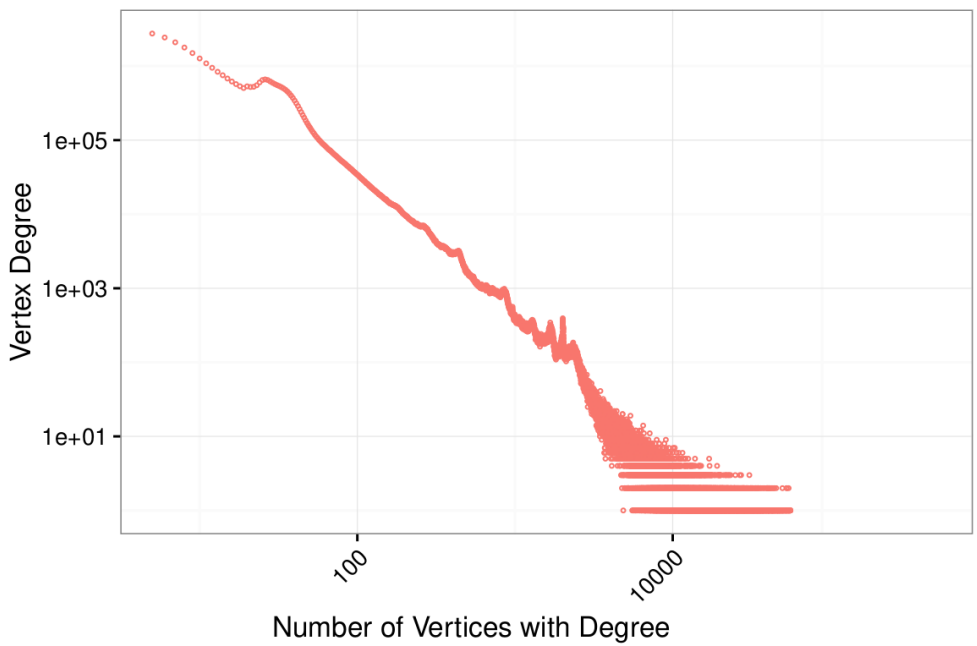
$$CC = \frac{\text{closed wedges}}{\text{all wedges}}$$

$$CC_v = \frac{5}{6}$$

Distribution Scaling

How to generate a 4× Twitter A-BTER graph

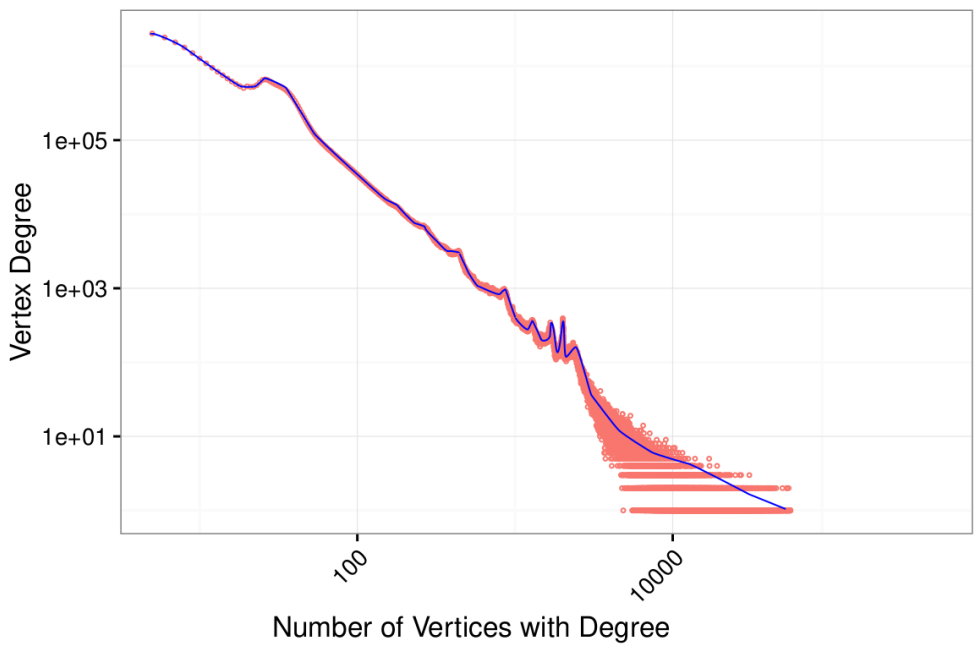
- Consider some input degree distribution (e.g., Twitter)



Distribution Scaling

How to generate a 4× Twitter A-BTER graph

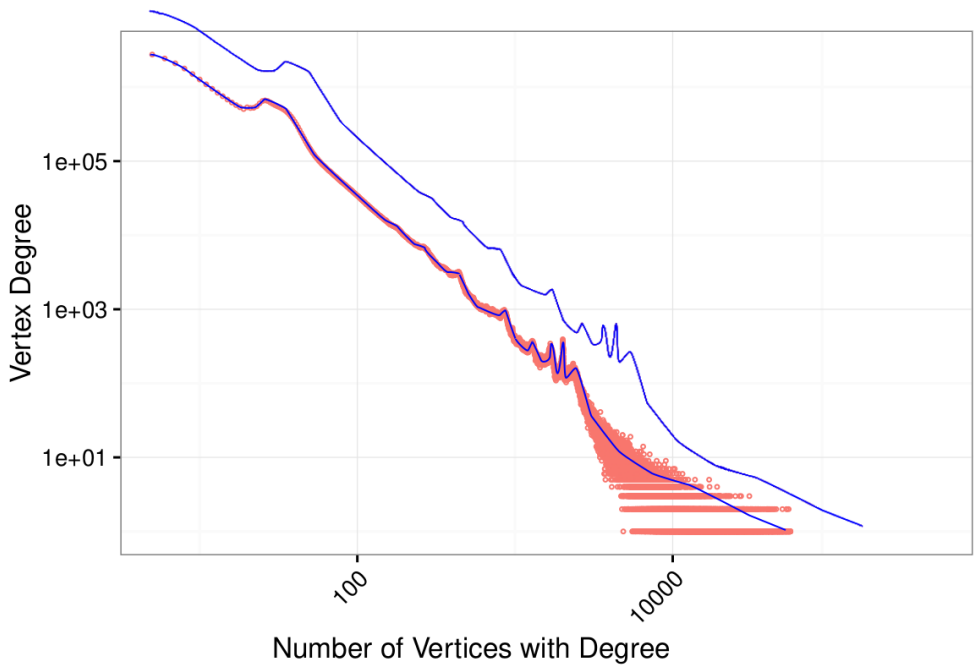
- Consider some input degree distribution (e.g., Twitter)
- We can interpolate and smooth this distribution to create a probability curve for degrees



Distribution Scaling

How to generate a 4× Twitter A-BTER graph

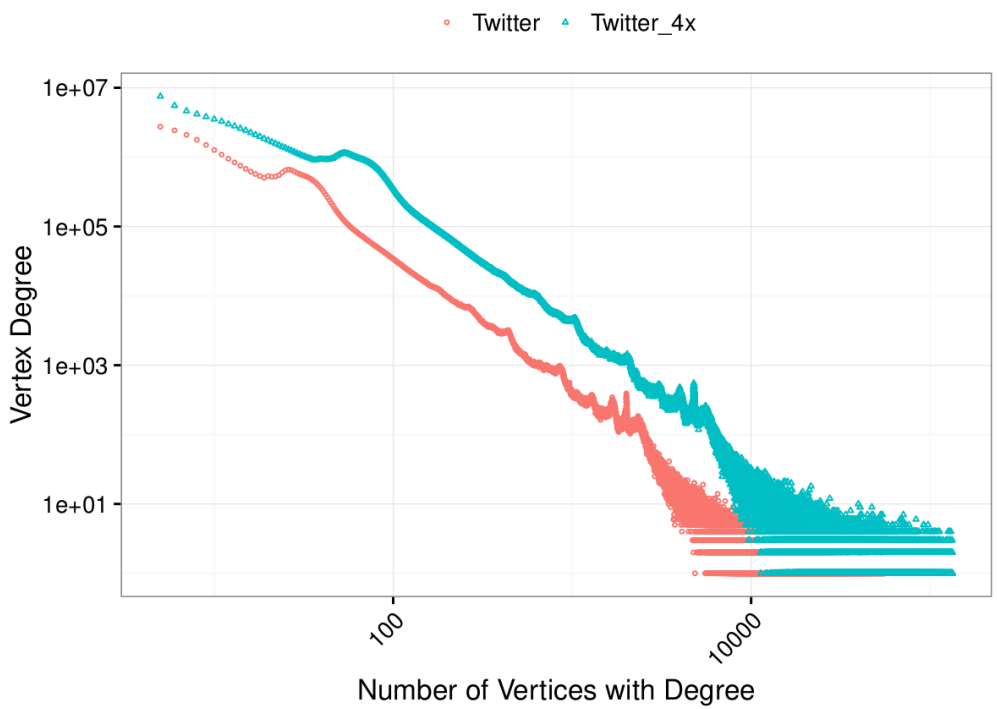
- Consider some input degree distribution (e.g., Twitter)
- We can interpolate and smooth this distribution to create a probability curve for degrees
- Then scale and shift this curve to analytically match some new # vertices and # edges



Distribution Scaling

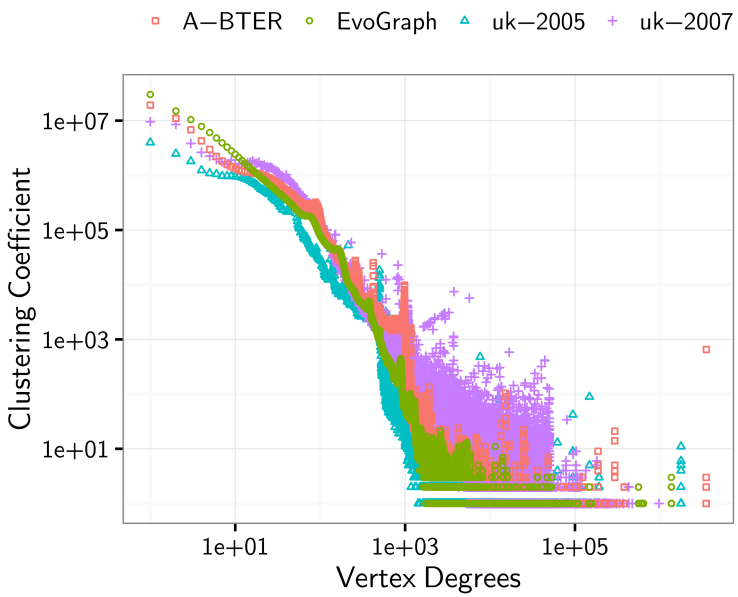
How to generate a 4× Twitter A-BTER graph

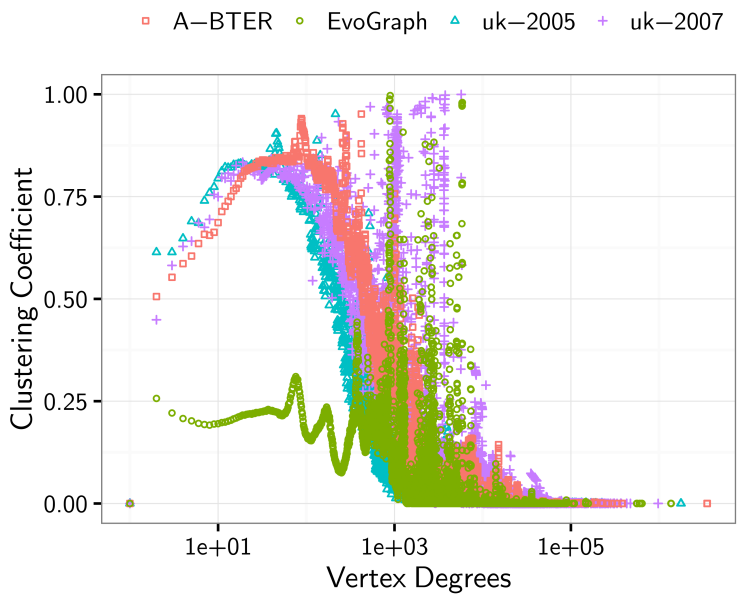
- Consider some input degree distribution (e.g., Twitter)
- We can interpolate and smooth this distribution to create a probability curve for degrees
- Then scale and shift this curve to analytically match some new # vertices and # edges
- Finally, we randomly sample using this curve "new # vertices" times to create the new distribution



Distribution Scaling
Comparison to real-world graph growth

- Our approach can often closely match real-world growth for degree and CC distributions
- We compare against uk-2005 and uk-2007 real world degree (left) and CC coefficients (right) as crawled from LAW[5]
- We scale from scaling uk-2005 to the N, M of uk-2007
  – Compare against graph growing EvoGraph [Park and Kim, 2018]





[5] Laboratory for Web Algorithmics

A-BTER: Adapted BTER

Our full approach for HPC-scale benchmark generation and evaluation

Input: Real or synthetic degree and clustering coefficient distributions

1. Optional step: Scale the input degree and clustering coefficient distributions
2. **Solve linear program to (further) shift clustering coefficient distributions to match some target μ_g**
3. Pass new distributions to an efficient edge-skipping based BTER implementation
4. Run community detection algorithm on generated graph, evaluate versus "engineered approximate solution" (EAS)

Output: A measure of algorithm solution quality

Linear Program - This paper has math!

Shifting the native μ of a graph's CC distribution

Minimally shift the input clustering coefficient (CC) distribution such that the output graph has a desired goal μ_g:

$$\mu_g = \frac{1}{N} \sum_{d}^{D} \frac{d_{inter}}{d} \qquad \textbf{definition of mixing parameter}$$

$$\text{minimize} \quad \sum_{d}^{D} |\hat{p}_d - p_d| \qquad \textbf{minimize shift in CC distribution}$$

$$\text{subject to} \quad \sum_{d}^{D} n_d \hat{p}_d = n(1 - \mu_g) \qquad \textbf{achieve target mixing parameter}$$

$$0 \leq \hat{p}_d \leq 1 \qquad \textbf{keep CC curve smooth and feasible}$$
$$|\hat{p}_d - p_d| \geq |\hat{p}_{d+1} - p_{d+1}|$$
$$|\hat{p}_d - \hat{p}_{d+1}| \leq 0.01$$

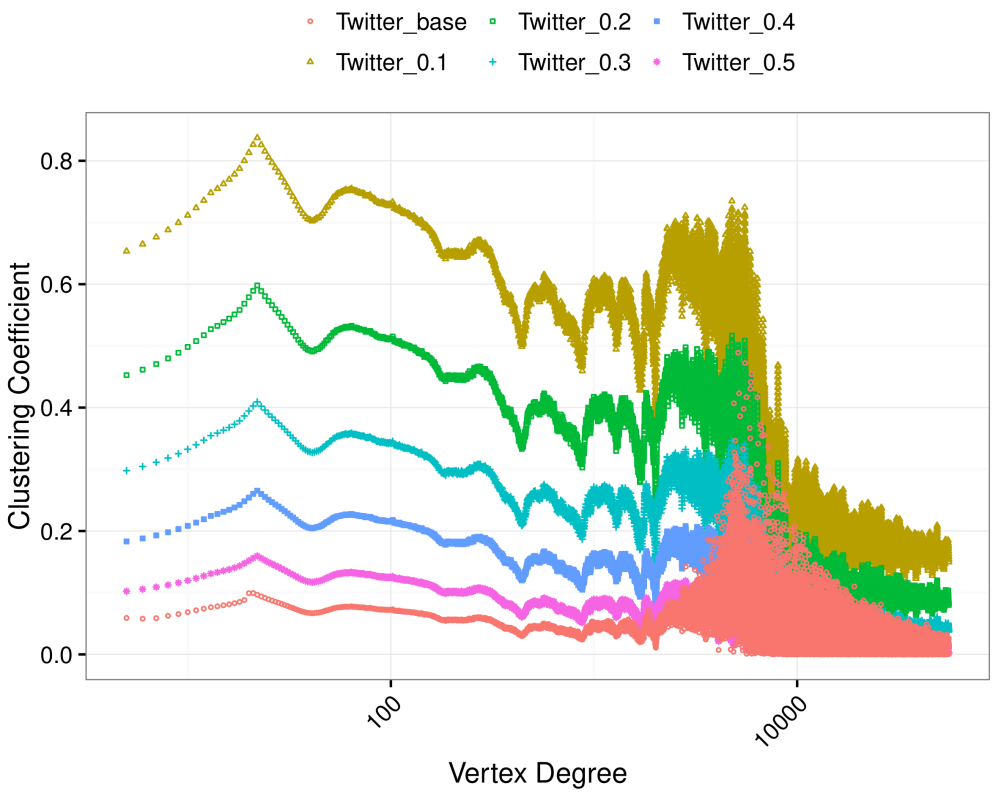
$$\text{output} \quad \hat{c}_d = \hat{p}_d^3$$

- p_d is $G(n,p)$ probabilities per degree from CC distribution c_d, $p_d = \sqrt[3]{c_d}$
- $\hat{p}_d$ is output probabilities to get new CC distribution $\hat{c}_d$, $\hat{c}_d = \hat{p}_d^3$
- n_d is degree distribution: n vertices of d degree – D is unique degrees
- d_{inter} is expected number of inter-community edges for vertex of degree d
- N is number of vertices in graph, M is number of edges

Shifting the CC distribution

CC distribution output running our LP on Twitter

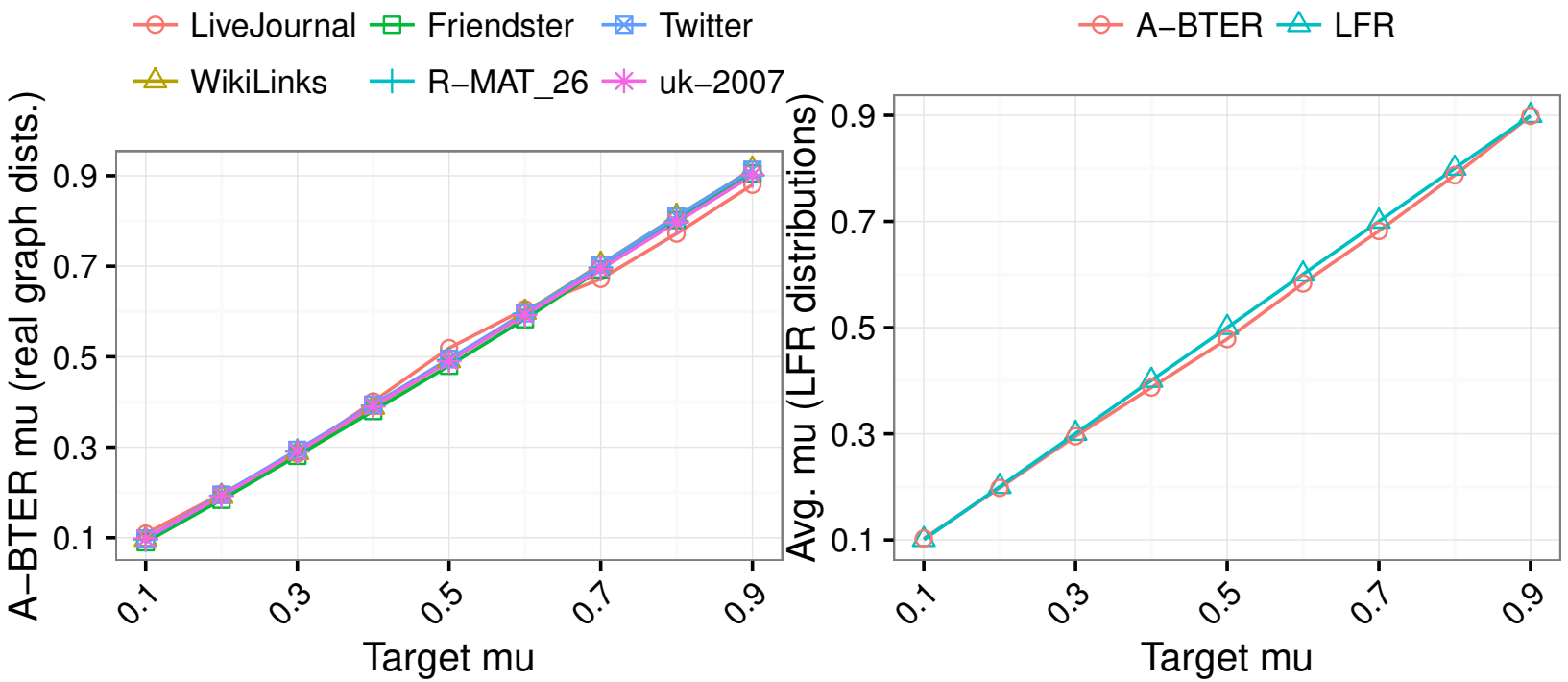
We can shift the CC distribution while still retaining *interesting* properties. We show the CC distributions for various μ on Twitter. Labeled as "Twitter_μ".



Generation Accuracy

Final outputs in terms of μ – TLDR: It works!

Our LP and CC shifting procedure has good accuracy in terms of achieving some target mixing parameter when combined with BTER. On the right, we compare to LFR itself by using distributions output by that generator.



Solution Time

Solve time for our LP isn't a deal-breaker

While LP solve times in general can be slow, we have several things working in our favor:

- We only require non-zeros in the degree distribution D to be variables in the LP
  – Generally, $|D| \ll d_{max} \ll M$
- We can utilize a binning strategy, where we group x vertices in degree order into a bin represented by a single variable in the LP
- Our most difficult test case (Twitter, ~20K nonzeros) takes **only a second to solve**

A-BTER: Adapted BTER

Our full approach for HPC-scale benchmark generation and evaluation

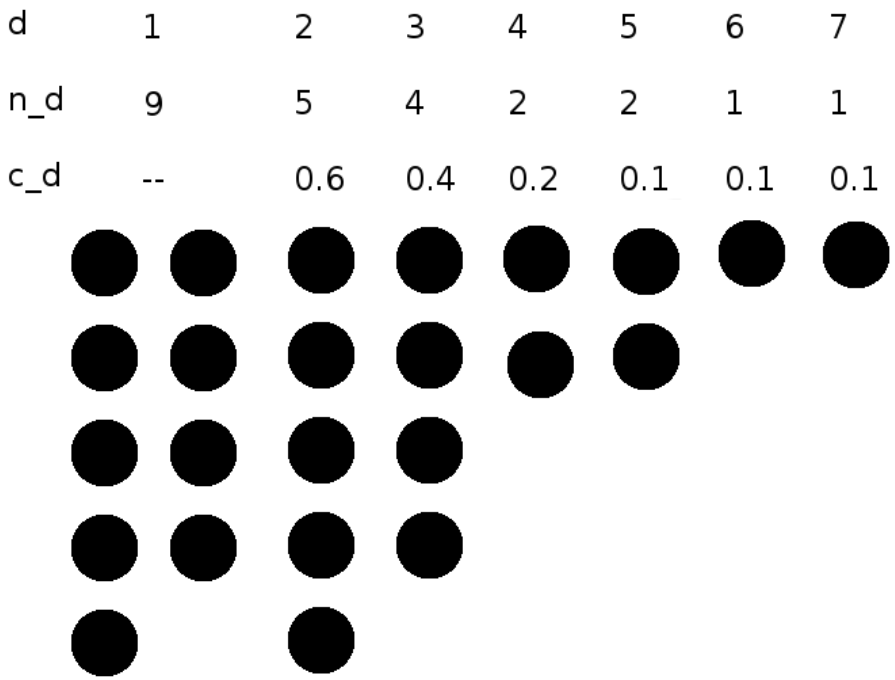
Input: Real or synthetic degree and clustering coefficient distributions

1. Optional step: Scale the input degree and clustering coefficient distributions
2. Solve linear program to (further) shift clustering coefficient distributions to match some target μ_g
3. **Pass new distributions to an efficient edge-skipping based BTER implementation**
4. Run community detection algorithm on generated graph, evaluate versus "engineered approximate solution" (EAS)

Output: A measure of algorithm solution quality

Background: BTER
Block Two-level Erdős-Rényi Graph Generator

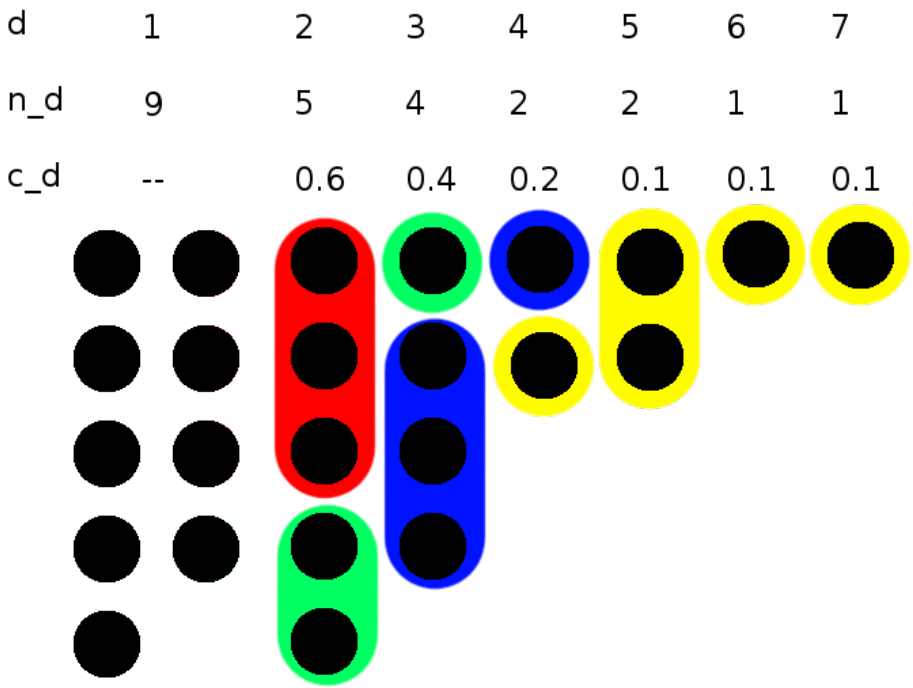
- Step 0: Input degree (n_d) and clustering coefficient (c_d) distributions



Background: BTER

Block Two-level Erdős-Rényi Graph Generator

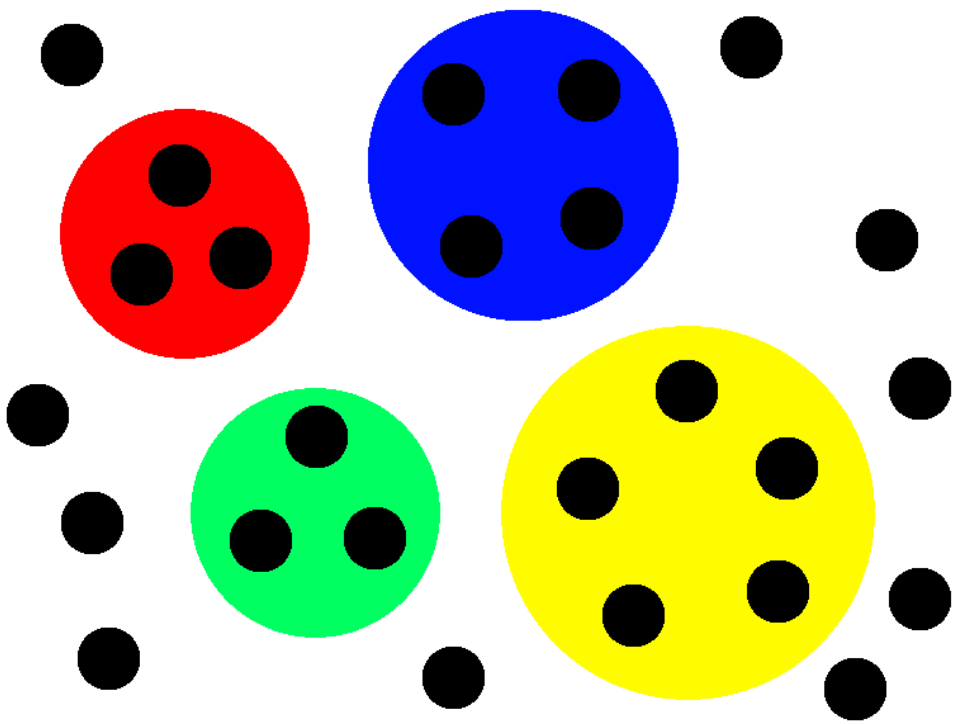
- Step 0: Input degree (n_d) and clustering coefficient (c_d) distributions
- Step 1: With ordered degree sequence, group $d+1$ vertices v of degree $d(v) >= d$ into *affinity blocks*



Background: BTER

Block Two-level Erdős-Rényi Graph Generator

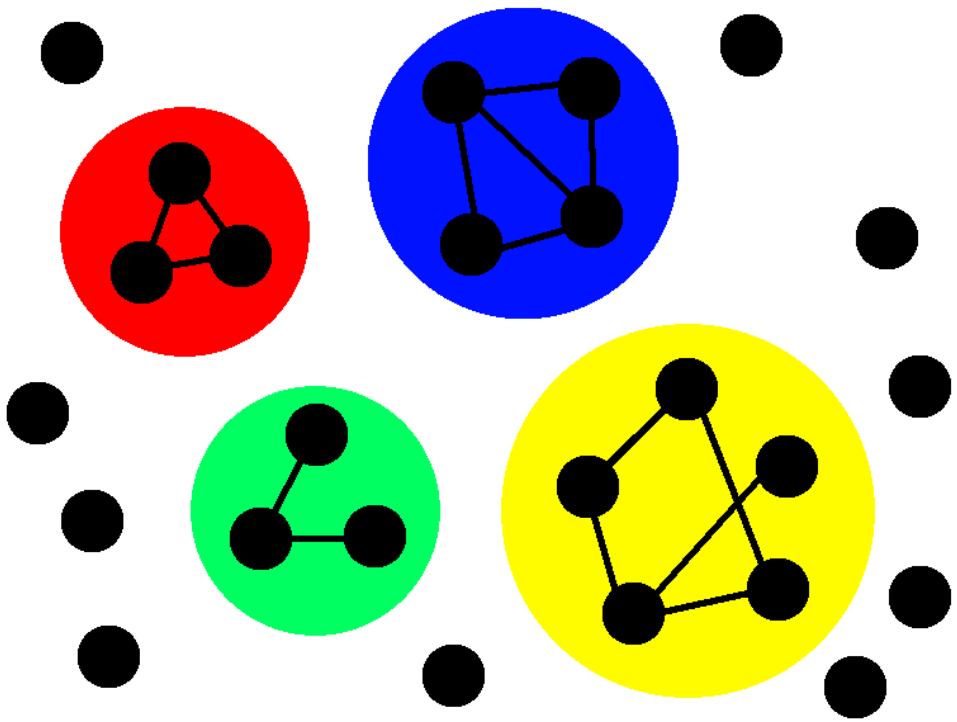
- Step 0: Input degree (n_d) and clustering coefficient (c_d) distributions
- Step 1: With ordered degree sequence, group $d+1$ vertices v of degree $d(v) >= d$ into *affinity blocks*



Background: BTER

Block Two-level Erdős-Rényi Graph Generator

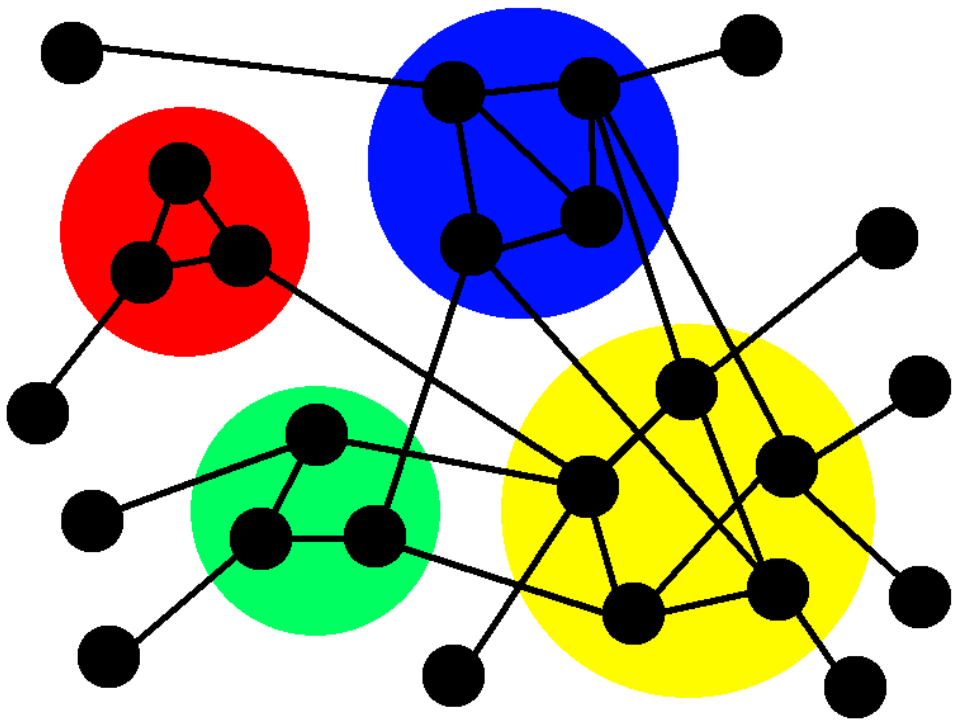
- Step 0: Input degree (n_d) and clustering coefficient (c_d) distributions
- Step 1: With ordered degree sequence, group $d+1$ vertices v of degree $d(v) >= d$ into *affinity blocks*
- Step 2: Use Erdös-Rényi probability $p_d = \sqrt[3]{c_d}$ to create intra-block edges via $G(n,p)$ process



Background: BTER

Block Two-level Erdős-Rényi Graph Generator

- Step 0: Input degree (n_d) and clustering coefficient (c_d) distributions
- Step 1: With ordered degree sequence, group $d+1$ vertices v of degree $d(v) >= d$ into *affinity blocks*
- Step 2: Use Erdös-Rényi probability $p_d = \sqrt[3]{c_d}$ to create intra-block edges via $G(n,p)$ process
- Step 3: Create inter-block edges via Chung-Lu process



Our Implementation - For Community Detection

How we adapt BTER for community detection benchmarking

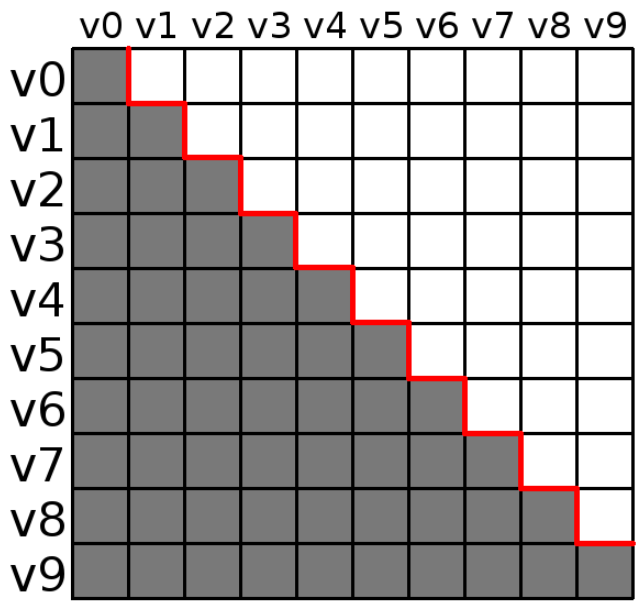
Adapting BTER:

- We wrap the BTER process to generate benchmark graphs
- Treat affinity blocks as EAS communities
- The LP shifting from the prior steps results in some goal mixing parameter μ_g based on EAS assignments
- We utilize the *edge-skipping* technique for very efficient generation

Background: Edge-skipping

For efficient Erdős-Rényi, Chung-Lu, and BTER graph generation

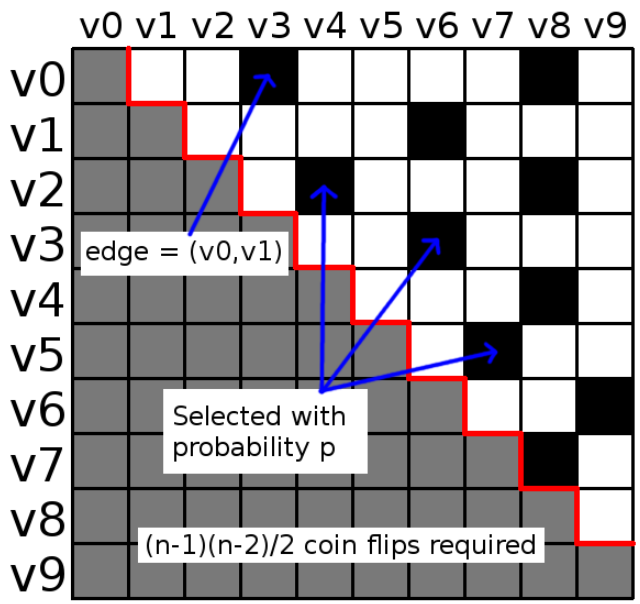
- Consider the undirected simple Erdős-Rényi $G(n, p)$ model as an adjacency matrix



Background: Edge-skipping

For efficient Erdős-Rényi, Chung-Lu, and BTER graph generation

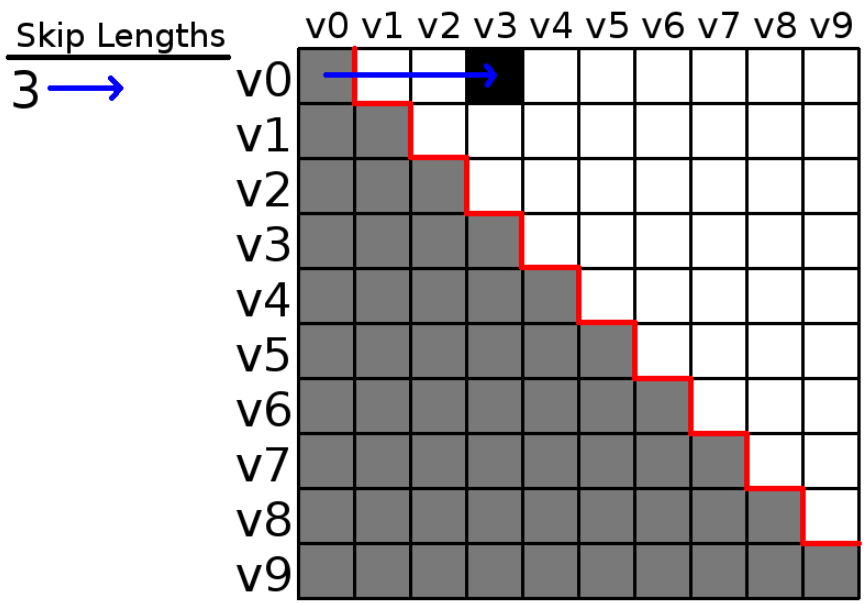
- Consider the undirected simple Erdős-Rényi $G(n, p)$ model as an adjacency matrix
- Nonzeros (i.e., edges) appear in upper triangle at index i, j with probability p



Background: Edge-skipping

For efficient Erdős-Rényi, Chung-Lu, and BTER graph generation

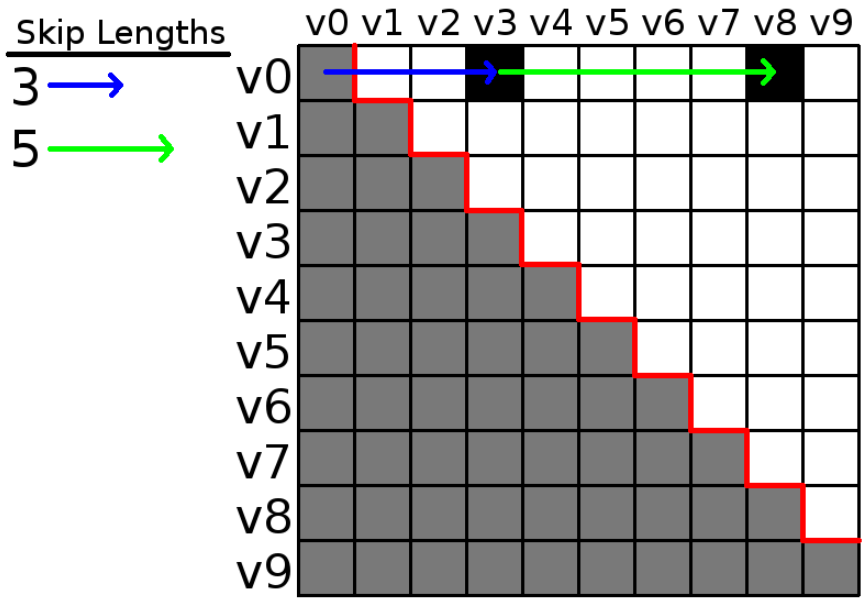
- Consider the undirected simple Erdős-Rényi $G(n, p)$ model as an adjacency matrix
- Nonzeros (i.e., edges) appear in upper triangle at index i, j with probability p
- Instead of flipping a coin for all indices, we can instead sample $\sim m = p \times \frac{(n-1)(n-2)}{2}$ *skip lengths*



Background: Edge-skipping

For efficient Erdős-Rényi, Chung-Lu, and BTER graph generation

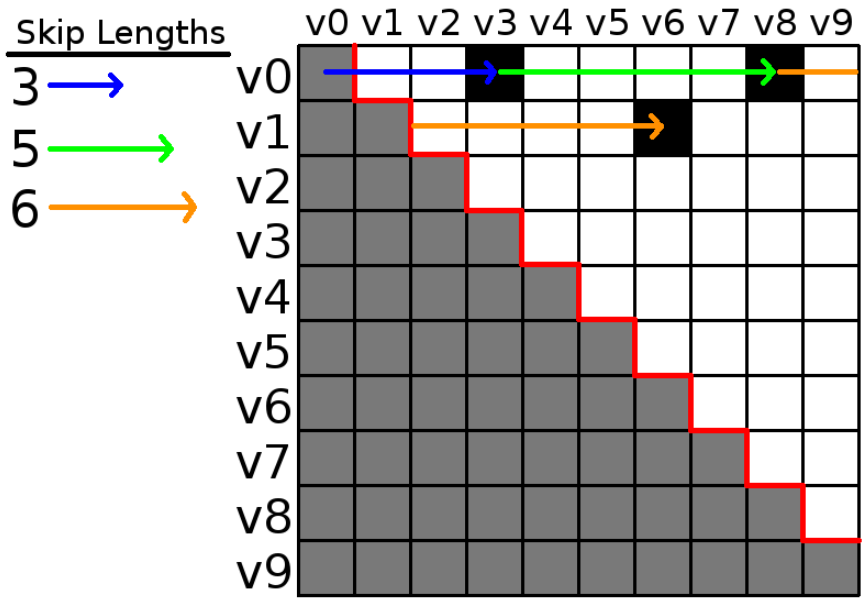
- Consider the undirected simple Erdős-Rényi $G(n, p)$ model as an adjacency matrix
- Nonzeros (i.e., edges) appear in upper triangle at index i, j with probability p
- Instead of flipping a coin for all indices, we can instead sample $\sim m = p \times \frac{(n-1)(n-2)}{2}$ *skip lengths*
- We traverse through an ordered space of unique possible edges, moving by each samples skip length and outputting the edge we land on



Background: Edge-skipping

For efficient Erdős-Rényi, Chung-Lu, and BTER graph generation

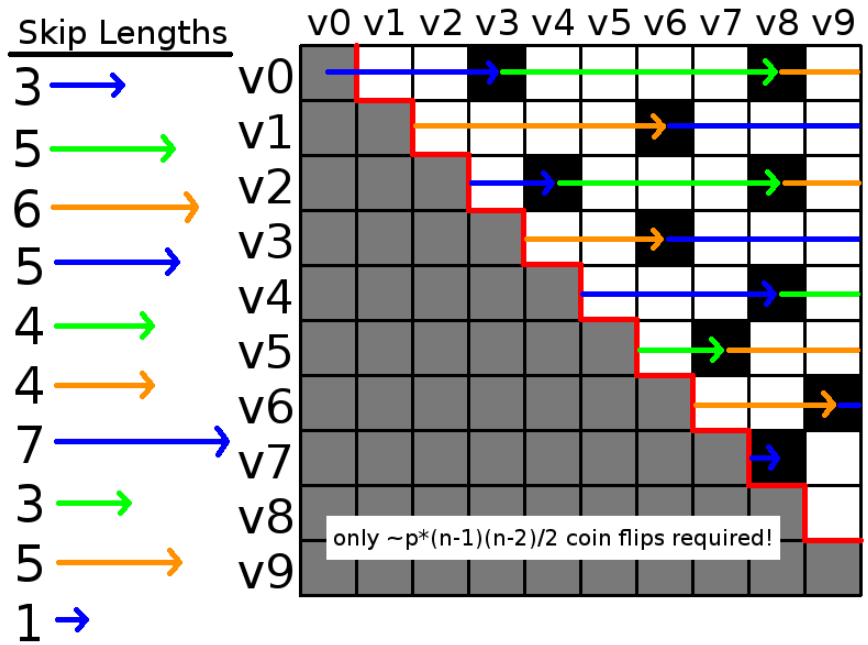
- Consider the undirected simple Erdős-Rényi $G(n, p)$ model as an adjacency matrix
- Nonzeros (i.e., edges) appear in upper triangle at index i, j with probability p
- Instead of flipping a coin for all indices, we can instead sample $\sim m = p \times \frac{(n-1)(n-2)}{2}$ *skip lengths*
- We traverse through an ordered space of unique possible edges, moving by each samples skip length and outputting the edge we land on



Background: Edge-skipping

For efficient Erdős-Rényi, Chung-Lu, and BTER graph generation

- Consider the undirected simple Erdős-Rényi $G(n, p)$ model as an adjacency matrix
- Nonzeros (i.e., edges) appear in upper triangle at index i, j with probability p
- Instead of flipping a coin for all indices, we can instead sample $\sim m = p \times \frac{(n-1)(n-2)}{2}$ *skip lengths*
- We traverse through an ordered space of unique possible edges, moving by each samples skip length and outputting the edge we land on



Edge-skipping

For Erdős-Rényi, Chung-Lu, and BTER graph generation

- Edge-skipping is provably equivalent to flipping a coin for each possible edge
- Recall: BTER creates Erdős-Rényi blocks and layers a Chung-Lu graph for inter-block edges
  - For Chung-Lu generation, we use edge skipping to create bipartite graphs with attachment probability $p_{i,j} = \frac{d_i \times d_j}{2m}$ for each unique degree pair (d_i, d_j)
  - Combining all of these bipartite graphs gives us the full Chung-Lu graph
  - **Combined with the Erdős-Rényi blocks, we have our BTER graph!**
- We can parallelize block and Chung-Lu generation with `MPI` and `OpenMP` to get a parallel time of $O(\frac{M}{P} + |D|)$

Edge Generation – Experimental setup for scaling

Non-graph SC people: you can start paying attention again

Test Systems:

- *Mutrino* – 96× KNL nodes with 68 cores, 96 GB DDR, 16 GB MCDRAM
- *Trinity* – 9800× KNL nodes with 68 cores, 96 GB DDR, and 16 GB MCDRAM
- *Astra* – 2500× ARM nodes with 56 cores and 128 GB DDR

Test Graphs:

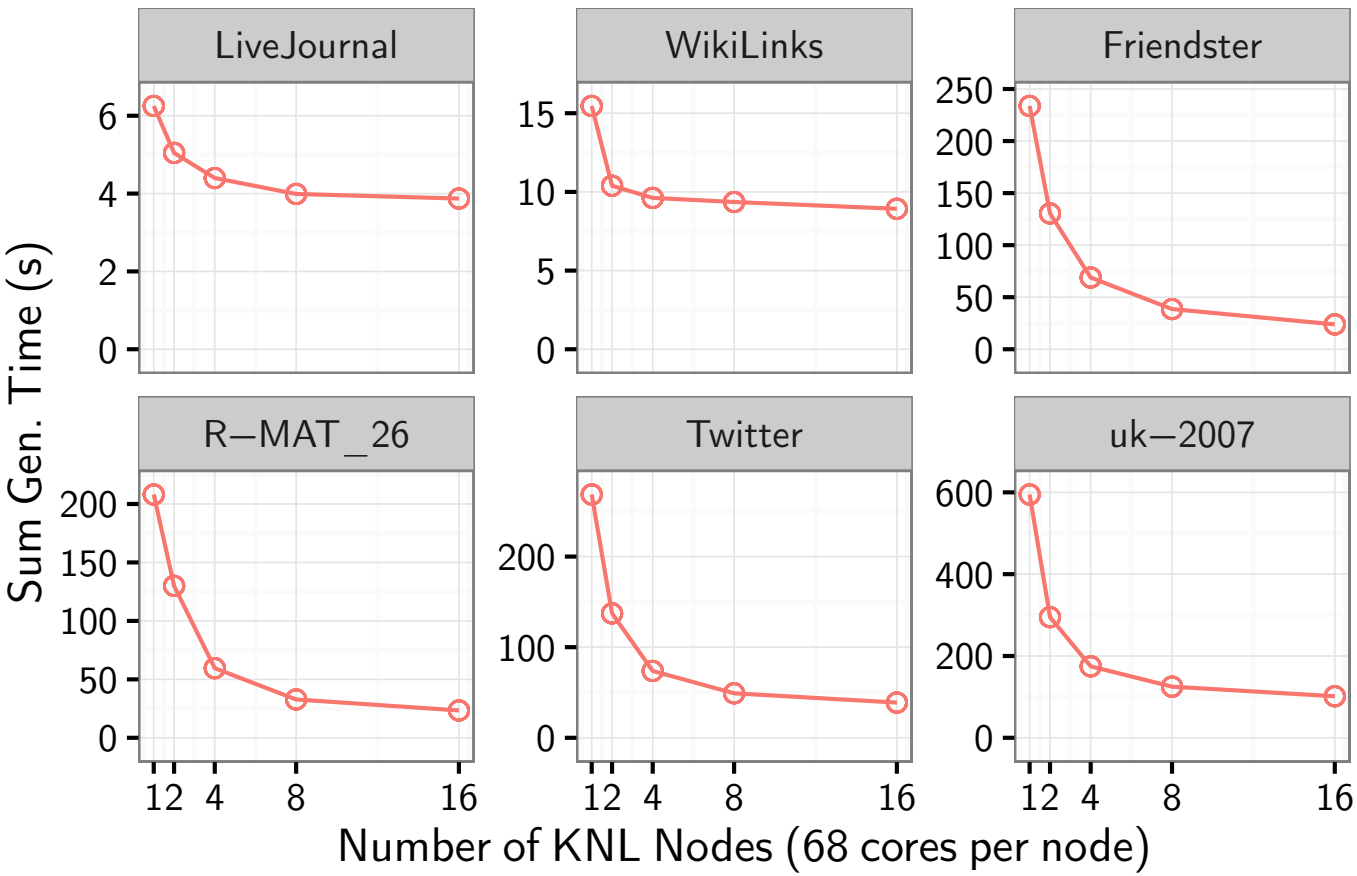
Network	N	M	d_{avg}	d_{max}	c_{avg}	c_{max}	Source
LiveJournal	2.1 M	25 M	24	2.0 K	0.27	0.39	SNAP
Wikilinks	1.9 M	21 M	21	8.6 K	0.12	0.18	Koblenz
RMAT_{26}	63 M	1.1 B	33	6.7 K	0.00	0.00	GTGraph
Twitter	39 M	1.4 B	73	56 K	0.07	0.49	Max Planck Inst.
Friendster	40 M	1.8 B	90	5.2 K	0.13	0.33	SNAP
uk-2007	81 M	3.3 B	80	82 K	0.78	0.99	LAW

We pre-process the distributions such that the minimum degree is 5 and maximum is $\sqrt{n}\log(n)$. This pre-processing is not necessary for our methods to work, but it enables more defined community boundaries.

Strong Scaling

Strong scalability of our edge-skipping BTER generator

- Strong scaling on *Mutrino* (KNL)
  - Generating 9 test graphs from $\mu = 0.1 \ldots 0.9$
- Average speedup is 5.8× across 16 nodes



Terascale Scaling

Scalability of our edge-skipping BTER generator

- Friendster scaled from 2× to 512× and generated on on *Trinity* (KNL) and *Astra* (ARM)
- The largest test utilizes 512 nodes of each system and generates a 15 terabyte edge list of 925 billion edges

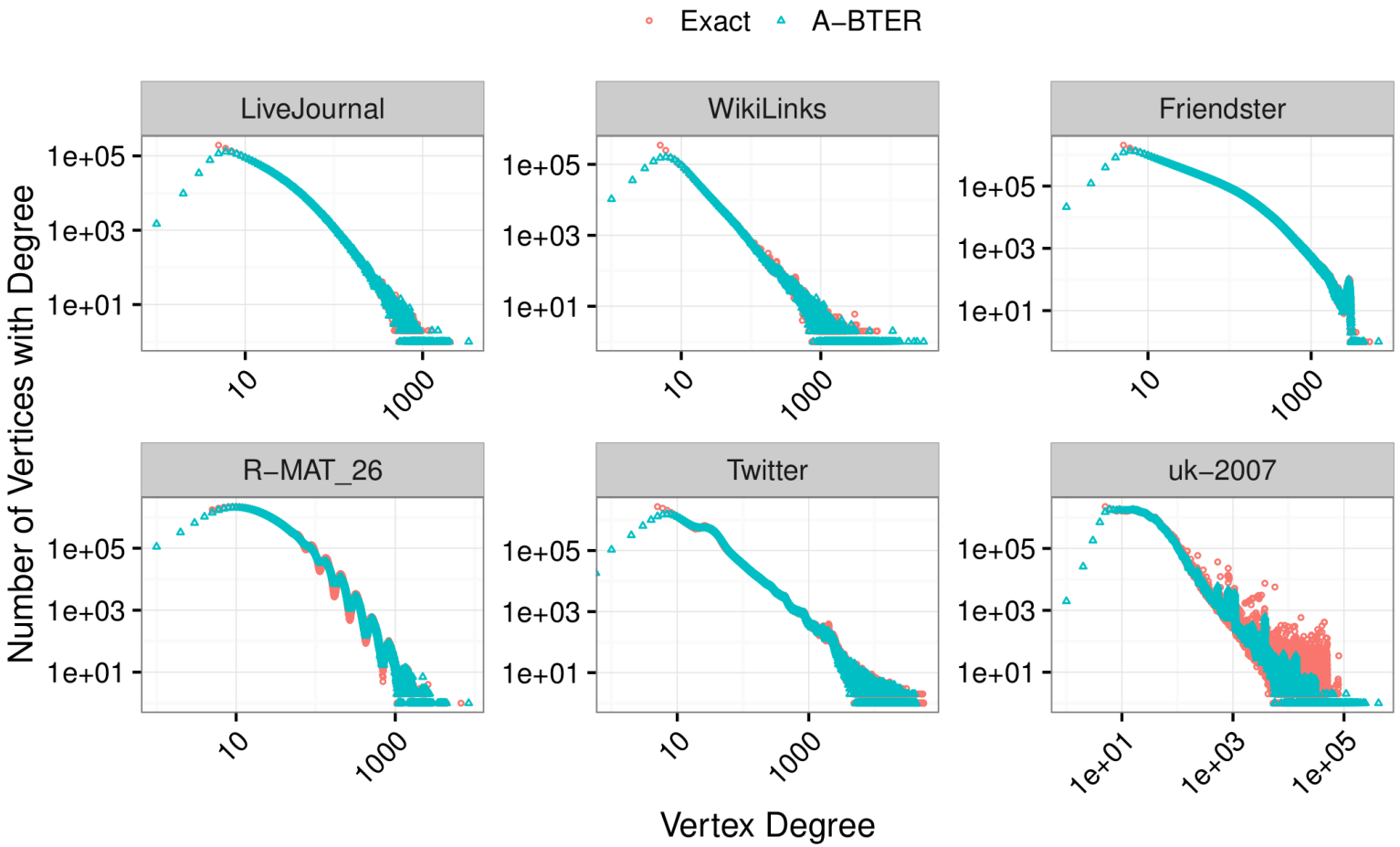
Scale	m	n	d_{max}	Memory	T_{KNL}	T_{ARM}
1×	1.8 B	40 M	5.2 K	29 GB	33 s	22 s
4×	7.2 B	93 M	10 K	115 GB	35 s	28 s
16×	29 B	260 M	15 K	459 GB	35 s	29 s
64×	115 B	786 M	20 K	1.8 TB	55 s	32 s
256×	464 B	2.5 B	26 K	7.4 TB	102 s	69 s
512×	**925 B**	4.6 B	30 K	15 TB	134 s	**76 s**

We generate edges at a rate of almost 1T per minute!

Matching Input Distributions

Output quality in terms of distribution matching

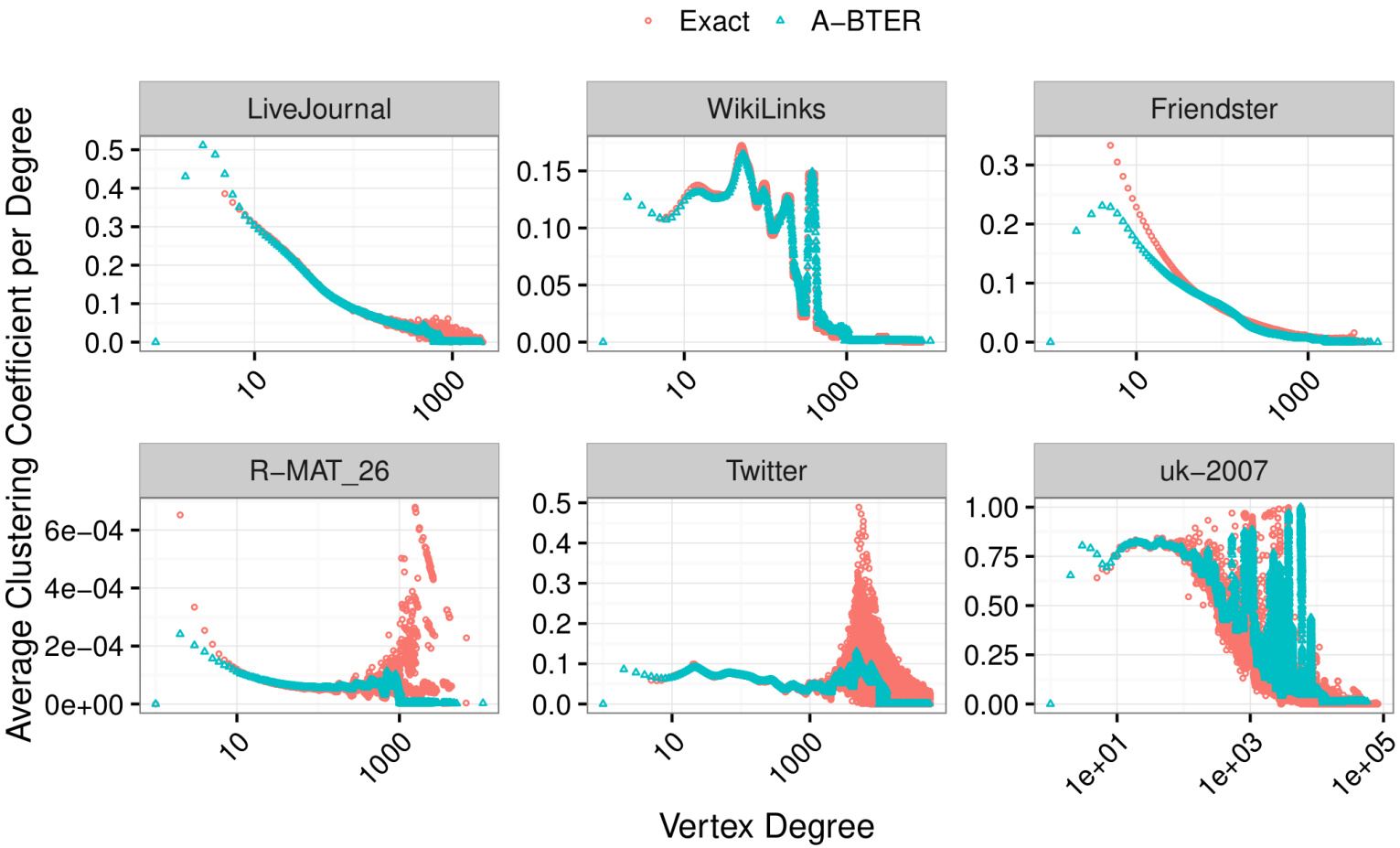
Our edge-skipping BTER generator also closely matches the input degree and clustering coefficient distributions.



Matching Input Distributions

Output quality in terms of distribution matching

Our edge-skipping BTER generator also closely matches the input degree and clustering coefficient distributions.



A-BTER: Adapted BTER

Our full approach for HPC-scale benchmark generation and evaluation

Input: Real or synthetic degree and clustering coefficient distributions

1. Optional step: Scale the input degree and clustering coefficient distributions
2. Solve linear program to (further) shift clustering coefficient distributions to match some target μ_g
3. Pass new distributions to an efficient edge-skipping based BTER implementation
4. **Run community detection algorithm on generated graph, evaluate versus "engineered approximate solution" (EAS)**

Output: A measure of algorithm solution quality

Using our Benchmark Generator

Benchmarking algorithms in practice

- We develop a parallel evaluation algorithm for Normalized Mutual Information
  - **Optimal work and parallel time complexity**
  - $O(n)$ and $O(1)$, respectively
  - Recent literature has claimed $O(n^2)$ work to compute
- We use it with A-BTER to benchmark parallel Louvain[6] and Label Propagation[7] community detection algorithms
  - And compare our benchmark performance vs. LFR
- We also benchmark Label Propagation at the very very large scale
  - And compare its output quality as we strong scale

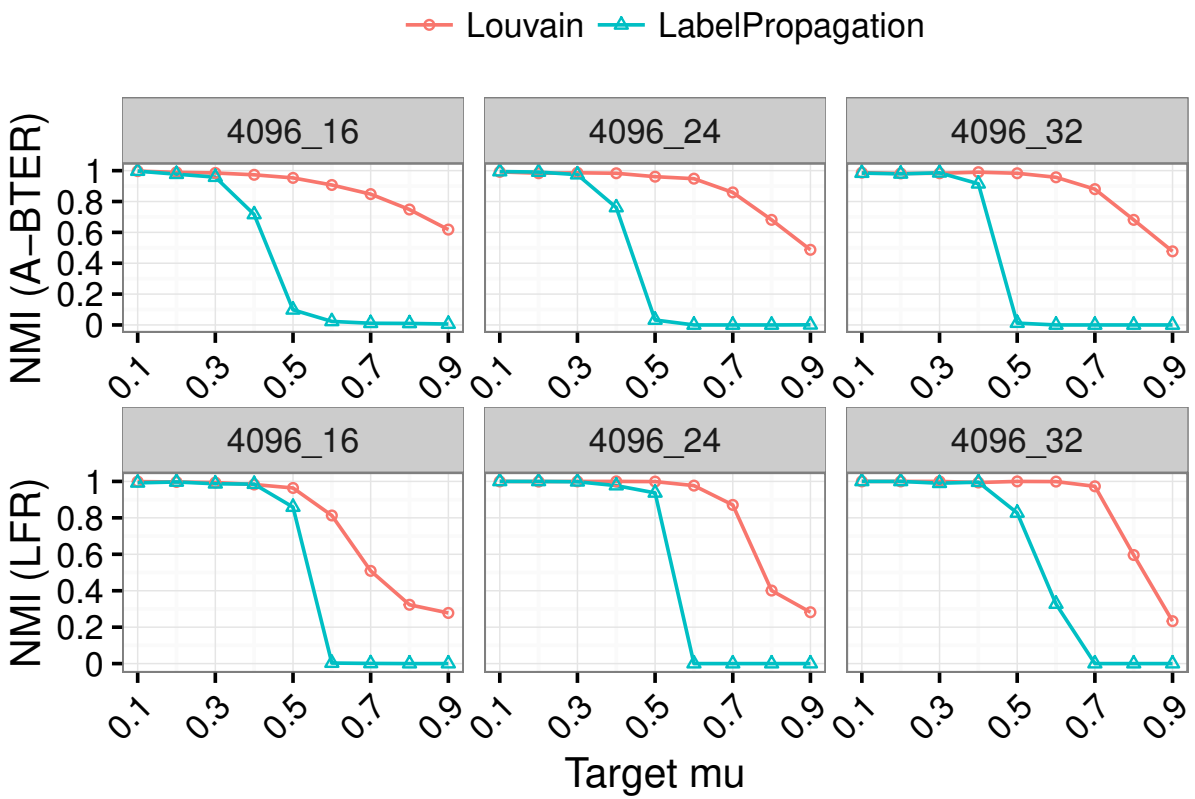
[6] [Ghosh et al., 2018]

[7] [Slota et al., 2016]

Benchmark Comparison to LFR

The conclusions drawn between benchmarks are the same

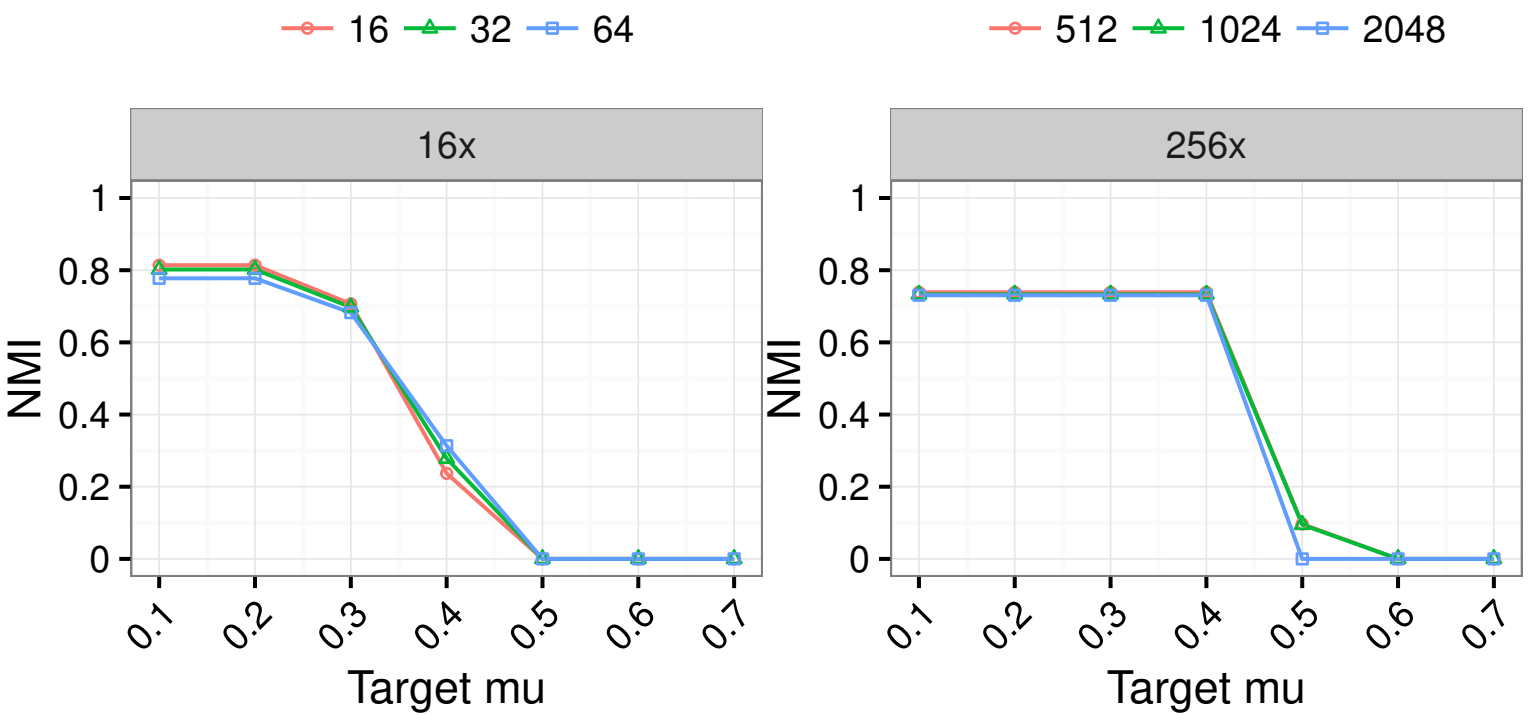
- We compare benchmark outputs from A-BTER (top) and LFR (bottom) – Read labels as (num vertices)_(average degree)
- We generate A-BTER graphs using degree and CC distributions from LFR
- We note similar observations in terms of performance of Louvain vs. Label Propagation – Takeaway: Louvain $>$ Label Propagation



Massive Scale Evaluation

Comparing algorithm output up to ~0.5T edges

- We run Label Propagation on 16× (left) and 256× (right) Friendster on various node counts
- We note running Label Propagation in distributed memory initially has a large effect on solution quality, though further strong scaling has minimal impact
- **First benchmark evaluation of community detection algorithms against Engineered Approximate Solution at HPC scale**



Conclusions
and future work

- Our approach can output graphs for community detection order-of-magnitudes faster than commonly-used generators, e.g., LFR
- Our approach can output graphs with more realistic degree and CC distributions than commonly-used generators
- We can scale up degree and CC distributions to generate terascale benchmarks
- Future Work: Develop generation methods for hierarchical or overlapping communities

If you propose a new community detection algorithm, you now have to use our code

Code: `www.github.com/HPCGraphAnalysis/SAGE`

`www.gmslota.com`, `slotag@rpi.edu`

Bibliography I

Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895. IEEE, 2018.

Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/o-efficient generation of massive graphs following the LFR benchmark. *J. Exp. Algorithmics*, 23(1):2.5:1–2.5:33, August 2018. ISSN 1084-6654. doi: 10.1145/3230743. URL http://doi.acm.org/10.1145/3230743.

Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, 2014.

Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4):1–5, October 2008. ISSN 1539-3755. doi: 10.1103/PhysRevE.78.046110. URL http://link.aps.org/doi/10.1103/PhysRevE.78.046110.

Joel C Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 115–126. Springer, 2011.

Himchan Park and Min-Soo Kim. Evograph: an effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2051–2059. ACM, 2018.

G. M. Slota, S. Rajamanickam, and K. Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.