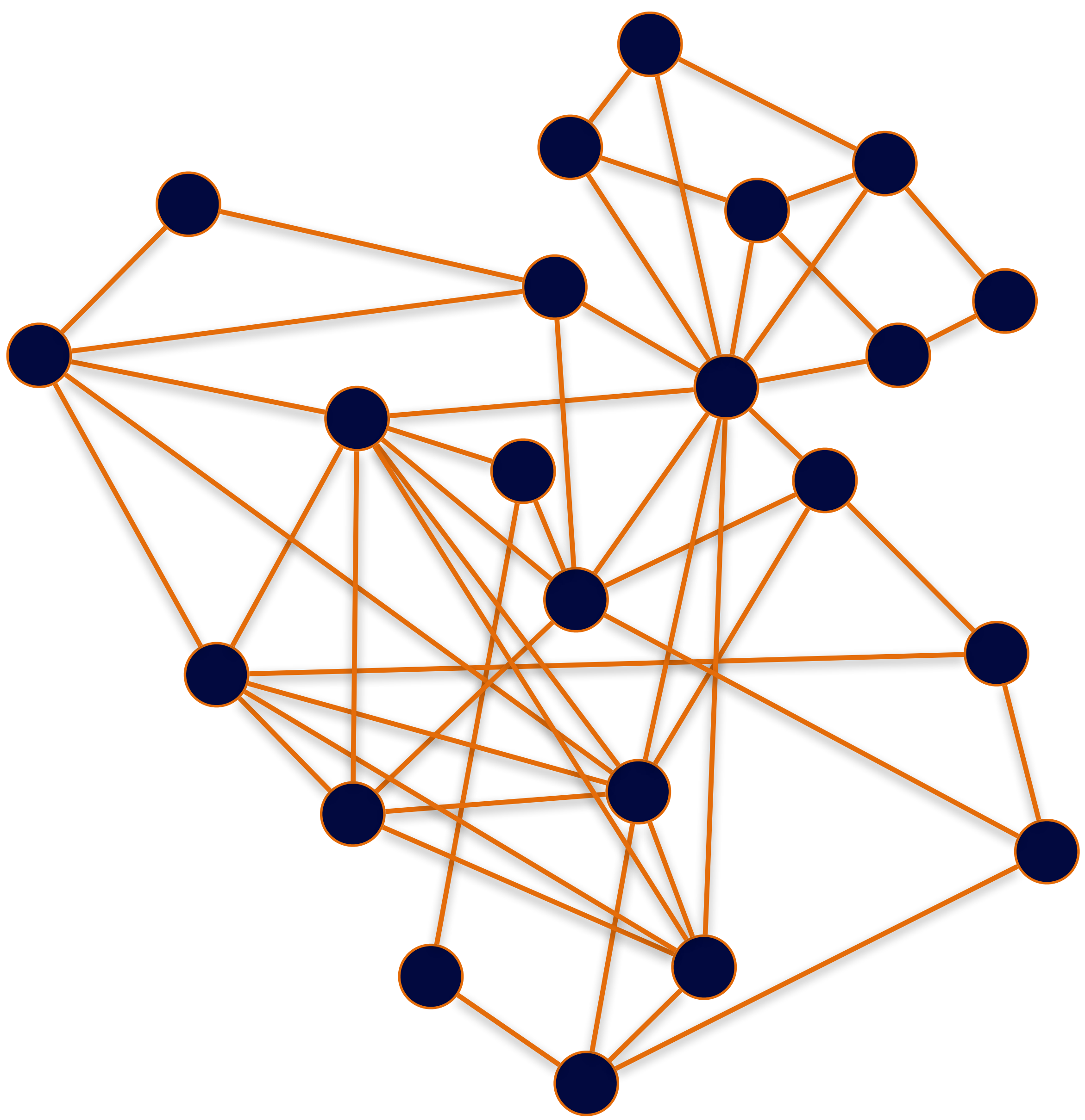
Finding Dense Subgraphs via Low-Rank Bilinear Optimization

Ioannis Mitliagkas

UT Austin

with: **Dimitris Papailiopoulos**
Alex Dimakis
Constantine Caramanis

Densest k-Subgraph (DkS)



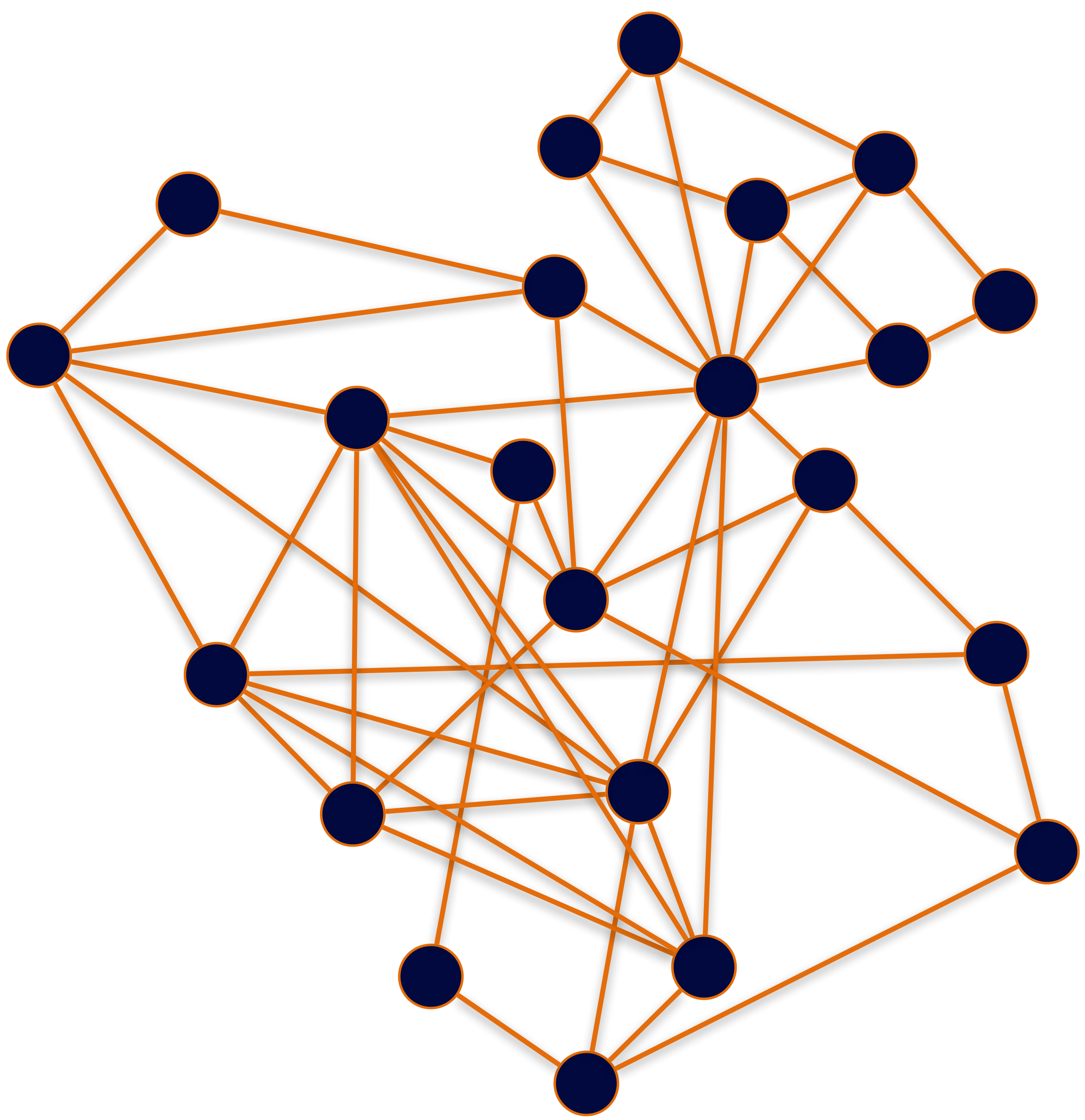
Given

graph and a **parameter k**

Find

k vertices containing **most edges**

Densest k-Subgraph (DkS)



Given

graph and a **parameter k**

Find

k vertices containing **most edges**

Applications

Community Mining

communities = large dense components

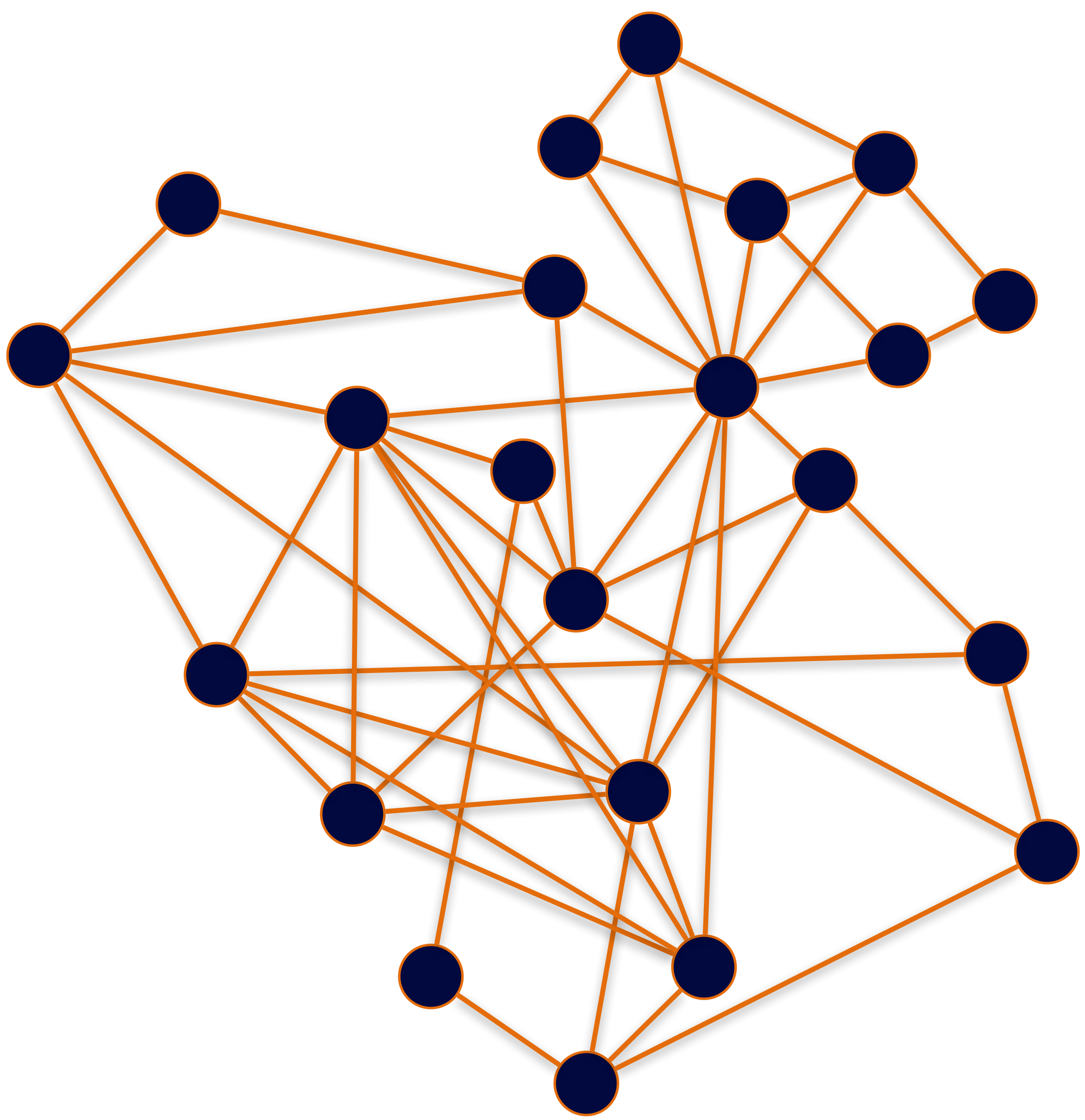
Link Spam Detection

dense parts of web: ***spam***

Computational biology

complex patterns in gene annotation graphs

Densest k-Subgraph (DkS)

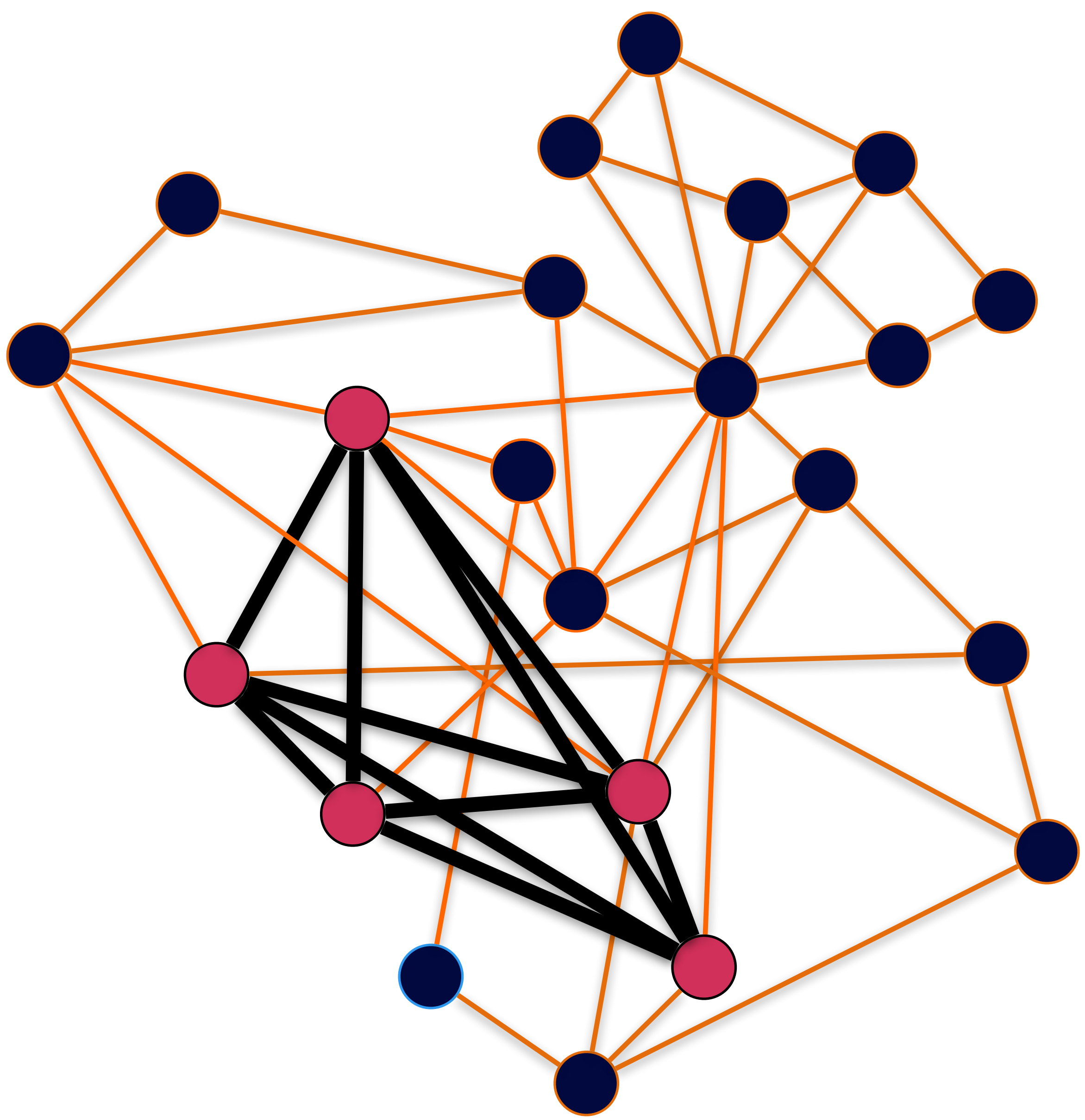


There is a

5-subgraph with 10 edges

Q: Can you find it?

Densest k-Subgraph (DkS)



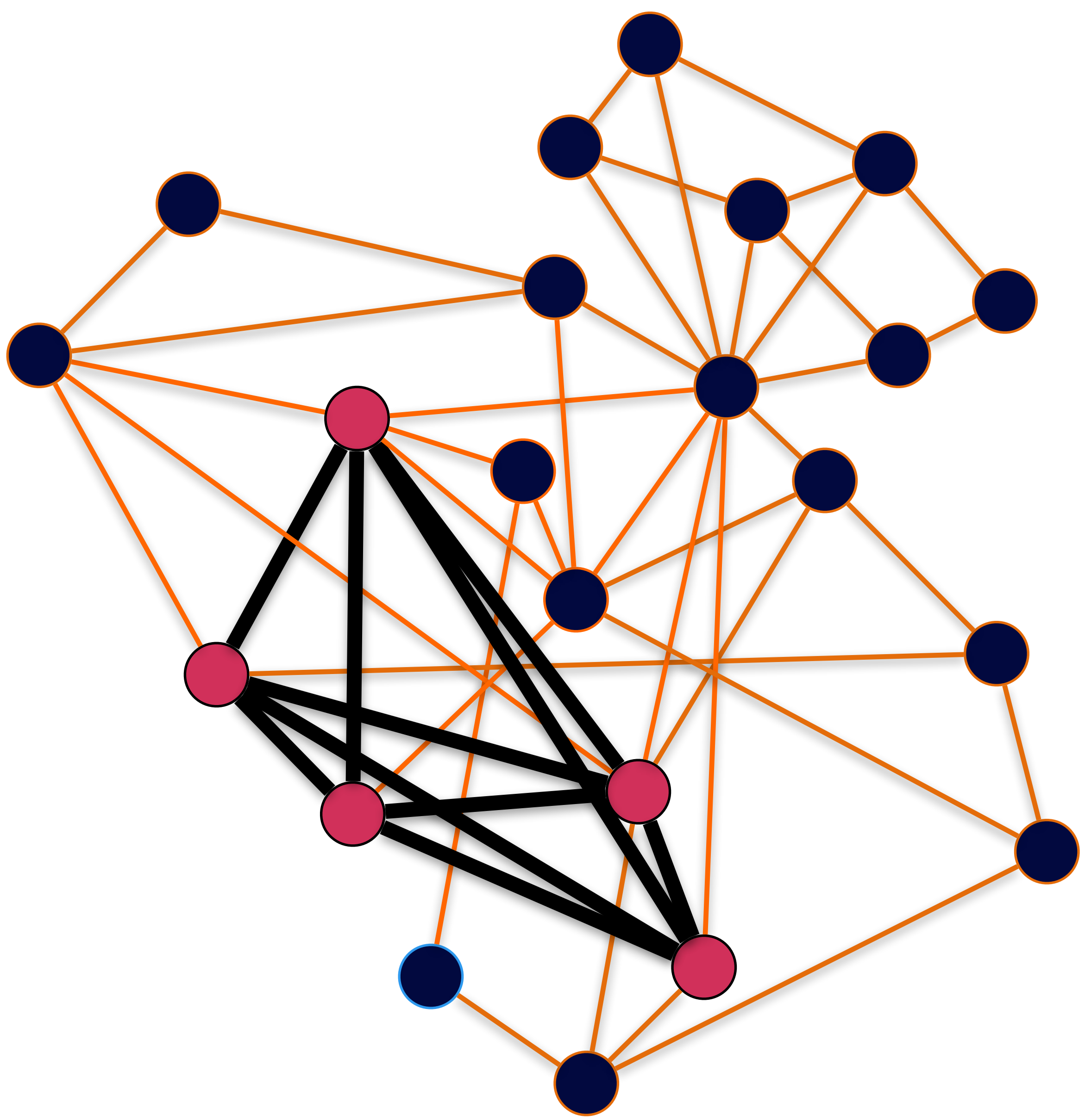
Given
graph and a **parameter k**

Find
k vertices containing **most edges**

NP-hard

Hard to approximate

Densest k-Subgraph (DkS)



Given
graph and a **parameter k**

Find
k vertices containing **most edges**

NP-hard

Hard to approximate

[Khot, 2004]

*Except in specific cases: [Arora et al 95]
(1+ε) approx. for linear subgraphs of dense graphs

Worst-Case Analysis

Worst-Case Analysis

$$\text{density} = \frac{2 \cdot \#\ \text{edges in subgraph}}{k} \qquad \text{(av.degree)}$$

$$\text{Approx} \geq \frac{\text{OPT}}{\rho}$$

Worst-Case Analysis

$$\text{density} = \frac{2 \cdot \#\ \text{edges in subgraph}}{k} \qquad \text{(av.degree)}$$

$$\text{Approx} \geq \frac{\text{OPT}}{\rho}$$

After long effort, [Feige, 2001], [Bhaskara et al., STOC ’10]

Best known **ratio**

$$\text{Approx} \geq \frac{\text{OPT}}{n^{0.25}}$$

10-factor approx. for graphs with **10K nodes**

100-factor approx. for graphs with **100 Million nodes**

Known DkS guarantees are not useful in practice…

under worst case analysis

Known DkS guarantees are not useful in practice…

under worst case analysis

Q1: Provable, graph-dependent bounds?

Q2: DkS on billion-scale graphs?

Beyond the Worst Case

New DkS algorithm:

Graph-dependent bounds

In practice: $\text{Approx} \geq 0.7 \cdot \text{OPT}$

Scalable

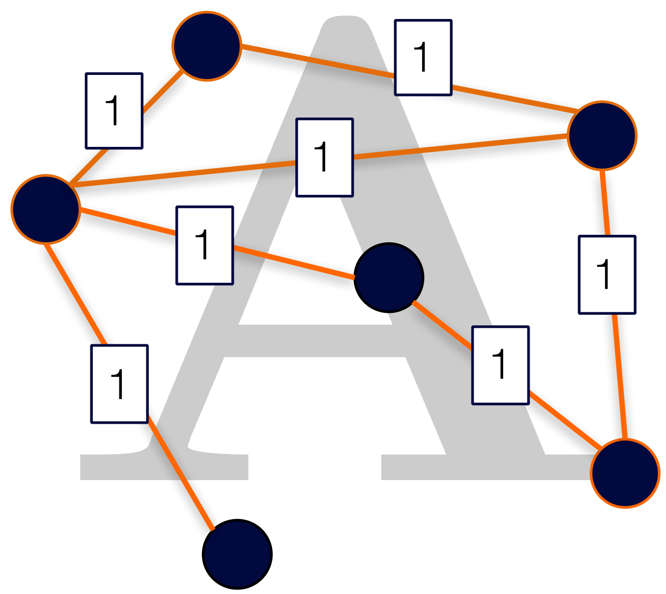
nearly-linear times for many real-world graphs

Parallelizable

implementation in **MapReduce+Python**

up to **billion-edge** graphs on **800 cores on Amazon EC2**

Our Low-Rank Framework



DkS on a graph
- Hard to solve
- Hard to approximate

Our Low-Rank Framework

DkS on a graph
- Hard to solve
- Hard to approximate

DkS on constant rank graph
- Nearly-linear time solvable (!)

Our Low-Rank Framework

DkS on a graph
- Hard to solve
- Hard to approximate

DkS on constant rank graph
- Nearly-linear time solvable (!)

Low-rank DkS is related to original DkS

Results: Theory

Graph-dependent Guarantees

$$\text{density} = \frac{2 \cdot \#\ \text{edges in subgraph}}{k} \qquad \text{(av.degree)}$$

Theorems:

Algorithm computes in **time O(n^{d+2}/δ)** a *k*-subgraph with **density**

$$\mathrm{OPT}_d \geq \mathrm{OPT} \cdot 0.5 \cdot (1-\delta) - 2|\lambda_{d+1}|$$

Graph-dependent Guarantees

$$\text{density} = \frac{2 \cdot \#\ \text{edges in subgraph}}{k} \qquad \text{(av.degree)}$$

Theorems:

Algorithm computes in **time O(n^{d+2}/δ)** a *k*-subgraph with **density**

$$\mathrm{OPT}_d \geq \mathrm{OPT} \cdot 0.5 \cdot (1 - \delta) - 2|\lambda_{d+1}|$$

If the **largest *d* eigenvalues** of the adjacency are **positive**

Our algorithm computes in **time** $O(|E| \cdot \log n + \frac{n}{\epsilon^d})$

a *k*-subgraph with **density**

$$\mathrm{OPT}_d \geq \mathrm{OPT} \cdot (1 - \epsilon) - 2|\lambda_{d+1}|$$

Graph-dependent Guarantees

$$\text{density} = \frac{2 \cdot \#\ \text{edges in subgraph}}{k} \qquad \text{(av.degree)}$$

Theorems:

Algorithm computes in **time O(n^{d+2}/δ)** a *k*-subgraph with **density**

$$\mathrm{OPT}_d \geq \mathrm{OPT} \cdot 0.5 \cdot (1 - \delta) - 2|\lambda_{d+1}|$$

If the **largest *d* eigenvalues** of the adjacency are **positive**

Our algorithm computes in **time** $O(|E| \cdot \log n + \frac{n}{\epsilon^d})$

a *k*-subgraph with **density**

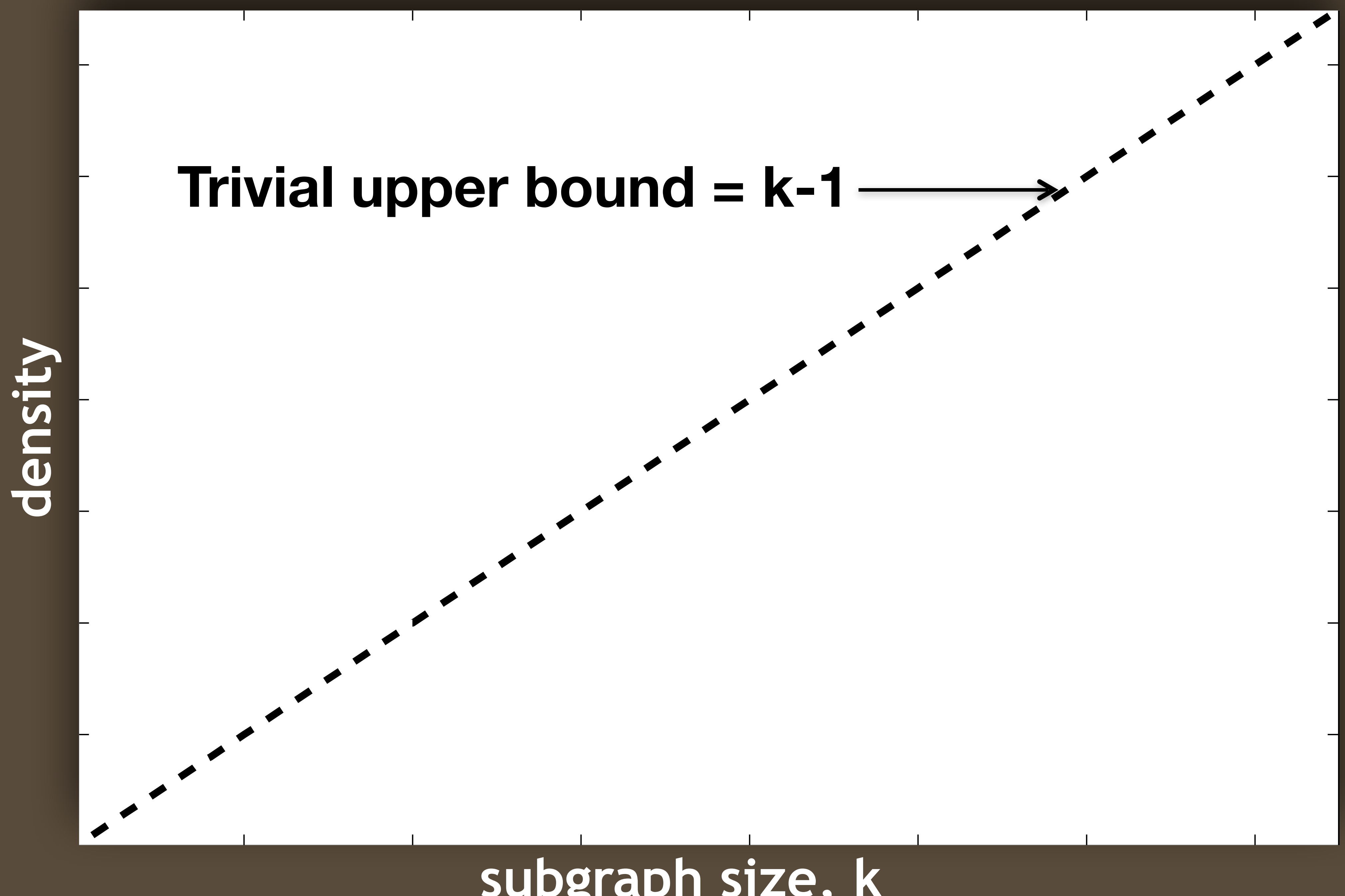
$$\mathrm{OPT}_d \geq \mathrm{OPT} \cdot (1 - \epsilon) - 2|\lambda_{d+1}|$$

larger *d* => **better** approximation, **slower** computation

Performance in Practice

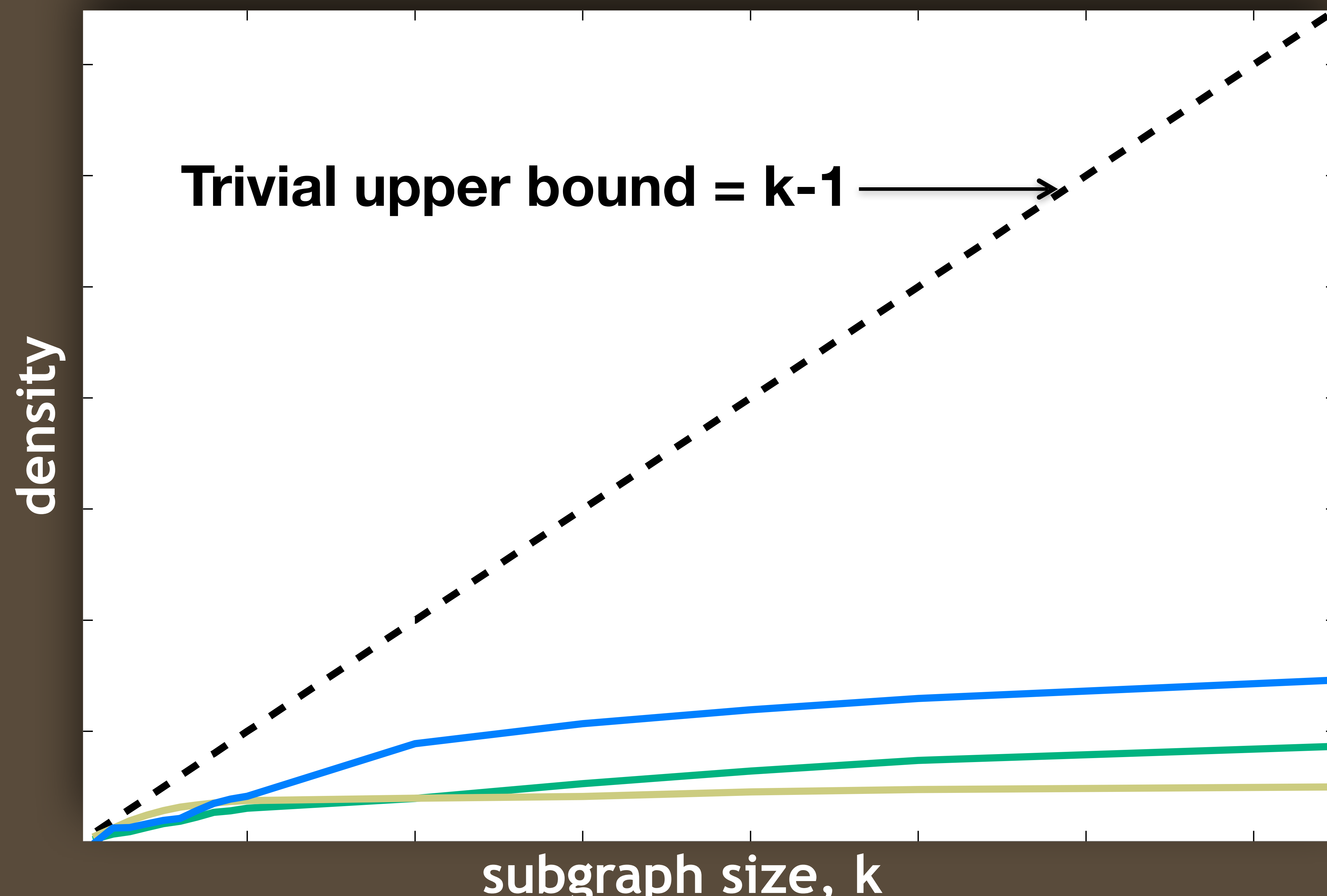
com-LiveJournal graph

4M nodes, 35M edges



com-LiveJournal graph

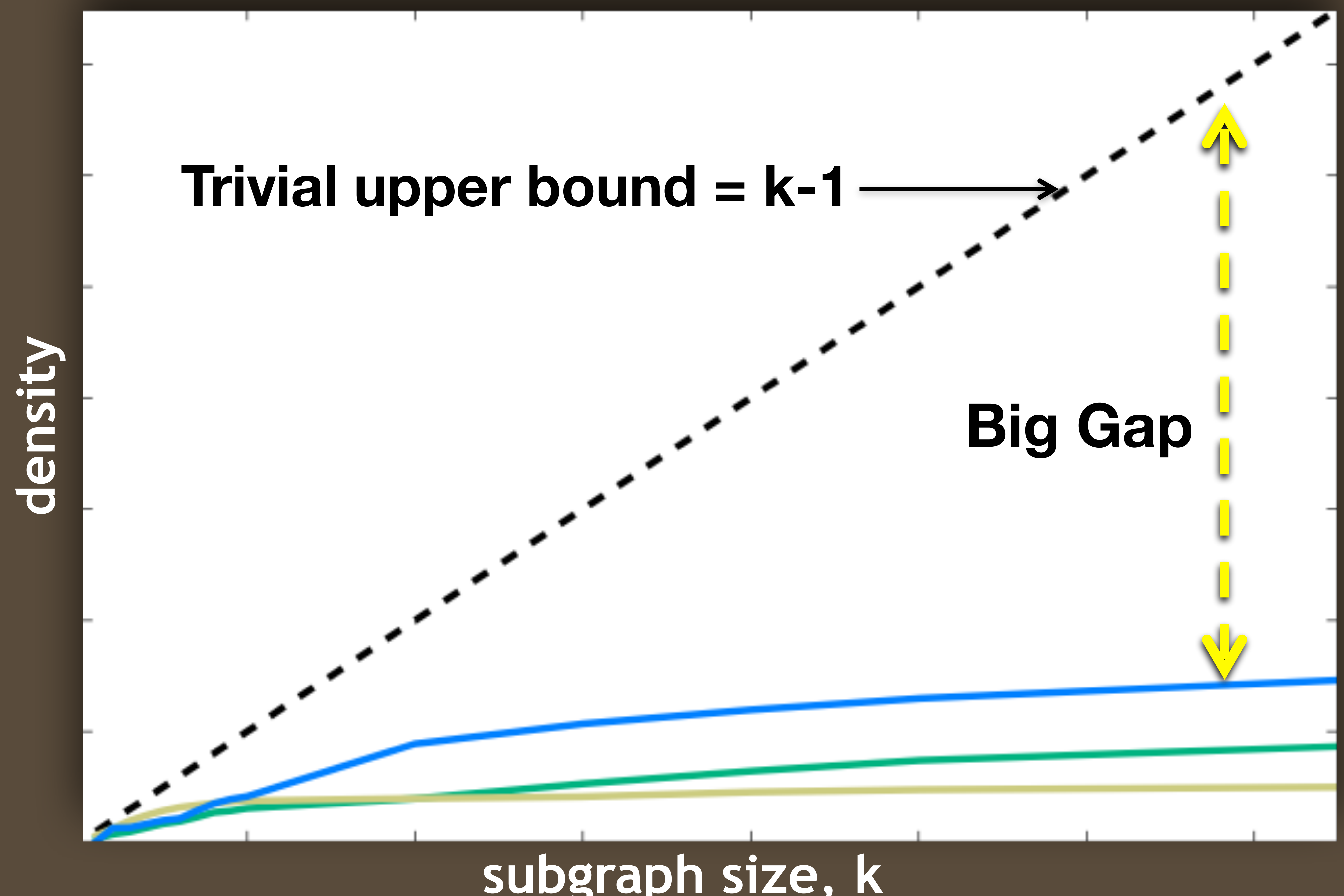
4M nodes, 35M edges



Blue: TPower JMLR'13 Green: GreedyFeige Algorithmica '01 Yellow: GreedyRavi OR'94

com-LiveJournal graph

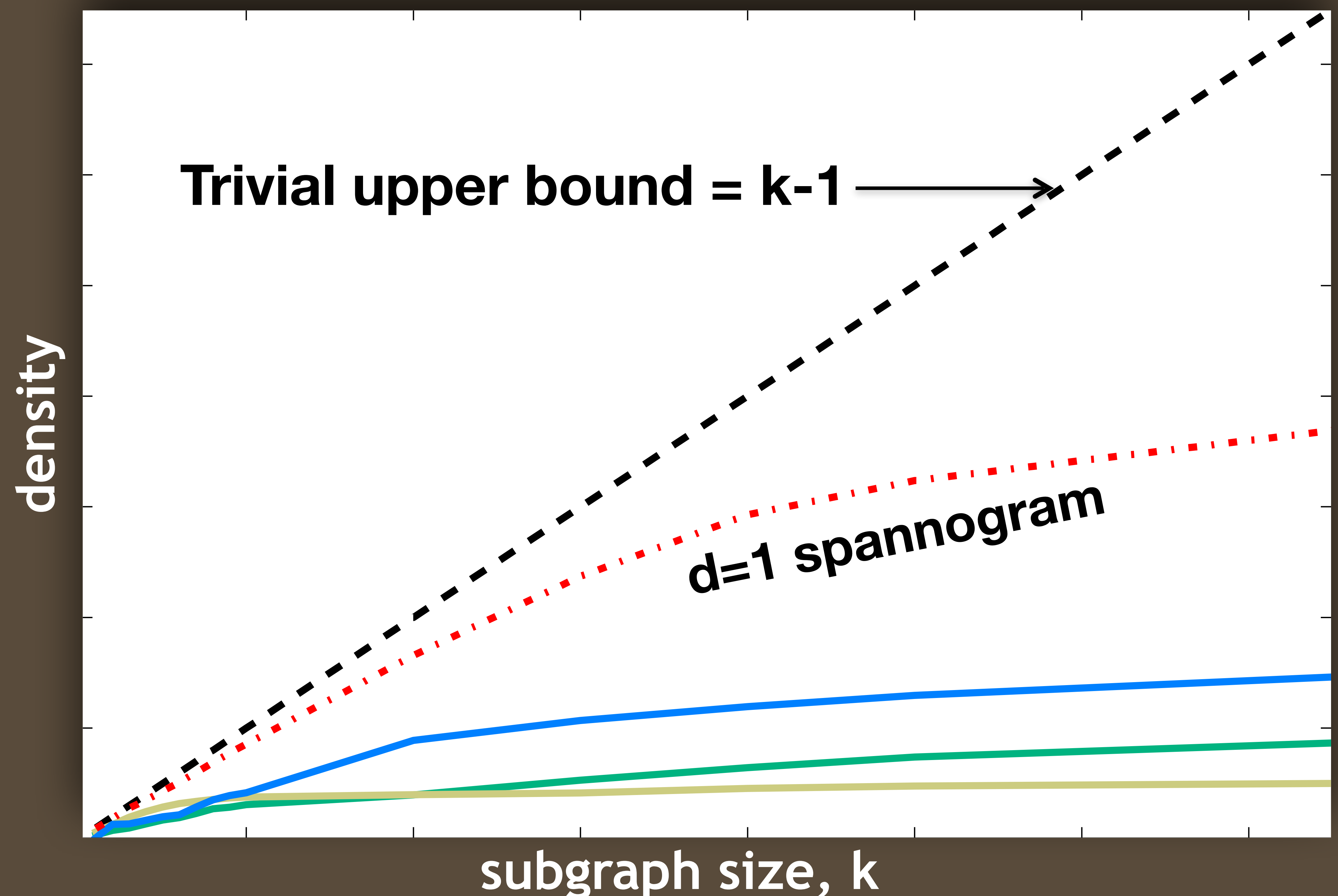
4M nodes, 35M edges



Blue: TPower JMLR'13 Green: GreedyFeige Algorithmica '01 Yellow: GreedyRavi OR'94

com-LiveJournal graph

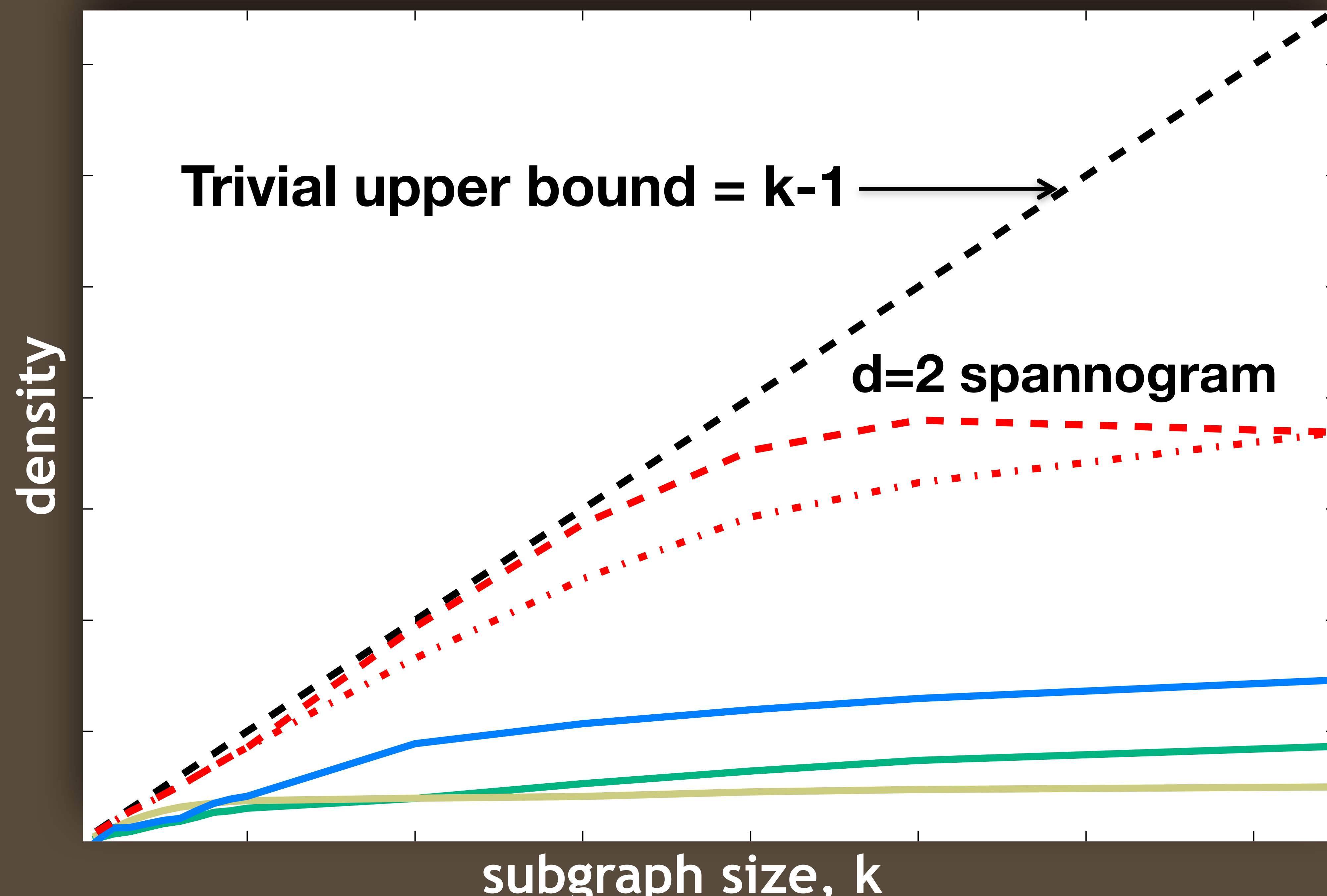
4M nodes, 35M edges



Blue: TPower JMLR'13 Green: GreedyFeige Algorithmica '01 Yellow: GreedyRavi OR'94

com-LiveJournal graph

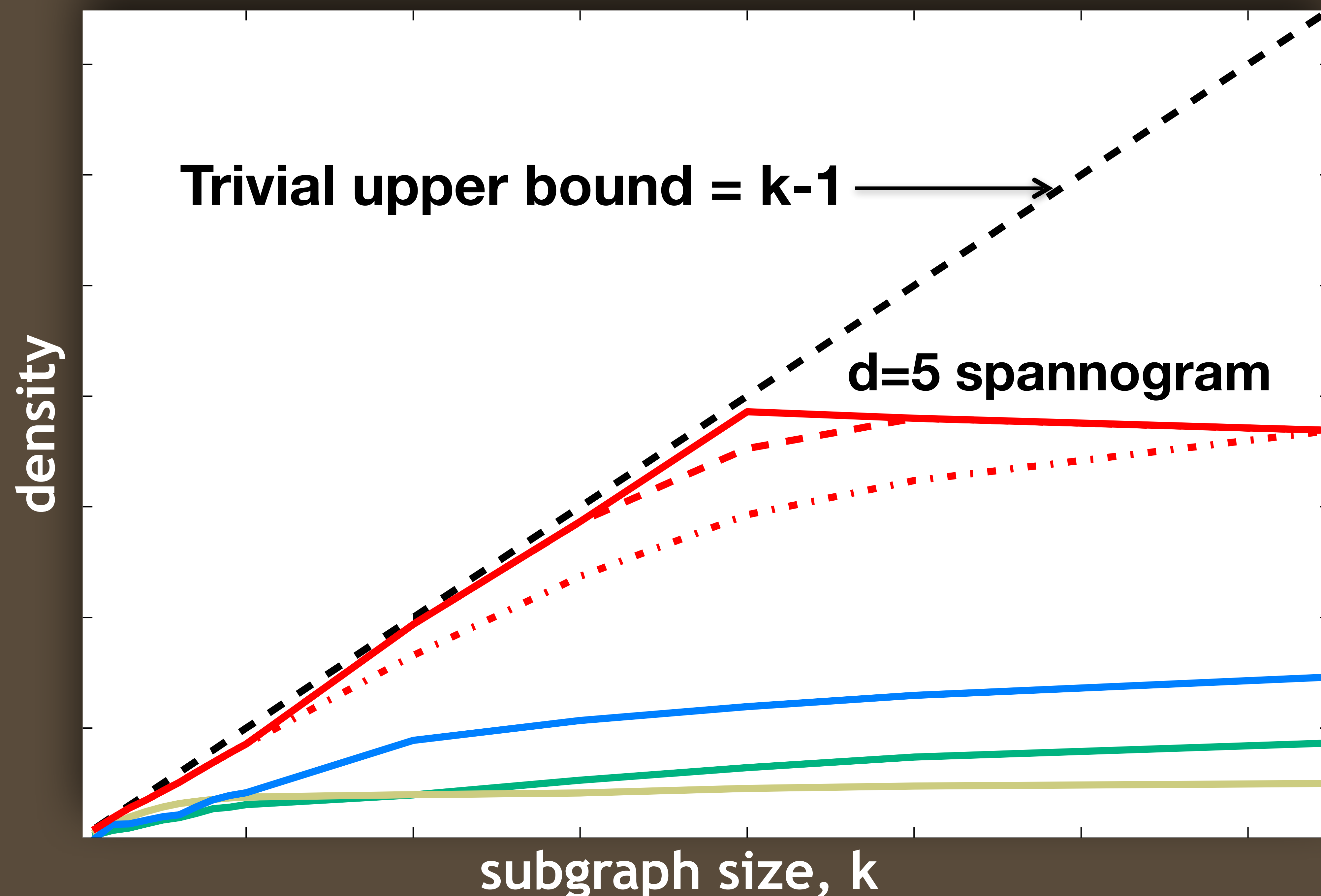
4M nodes, 35M edges



Blue: TPower JMLR’13 Green: GreedyFeige Algorithmica ’01 Yellow: GreedyRavi OR’94

com-LiveJournal graph

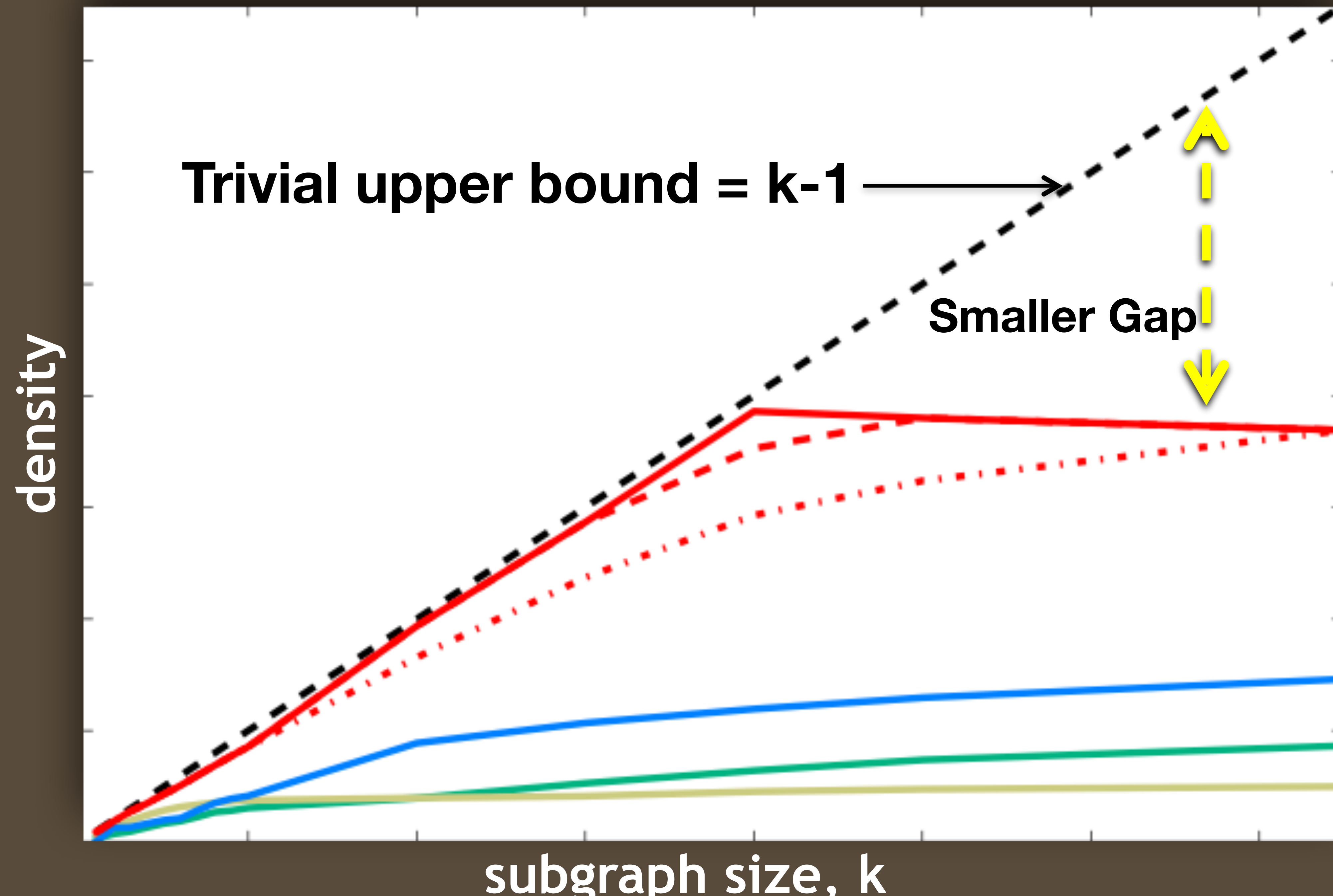
4M nodes, 35M edges



Blue: TPower JMLR'13 Green: GreedyFeige Algorithmica '01 Yellow: GreedyRavi OR'94

com-LiveJournal graph

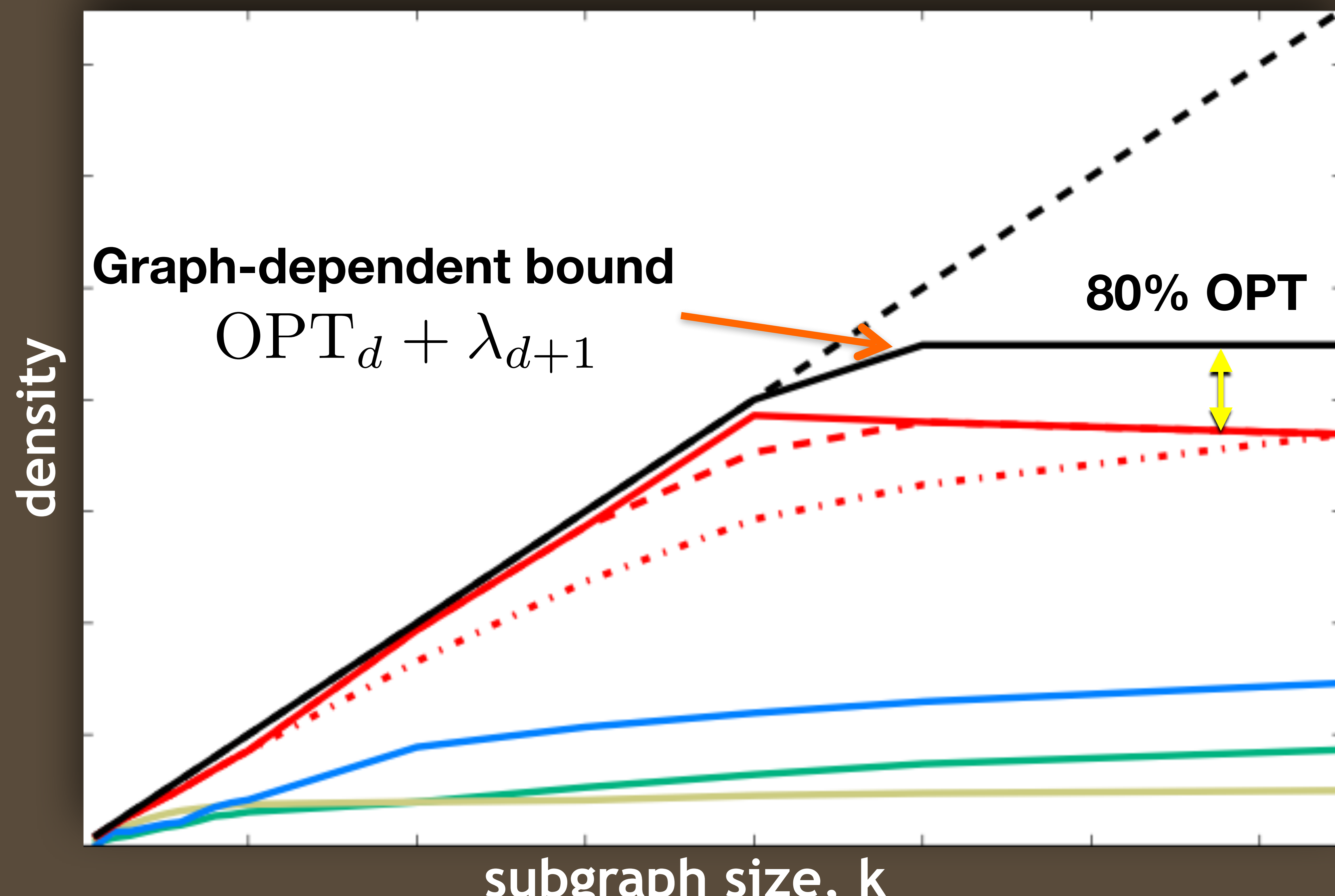
4M nodes, 35M edges



Blue: TPower JMLR'13 Green: GreedyFeige Algorithmica '01 Yellow: GreedyRavi OR'94

com-LiveJournal graph

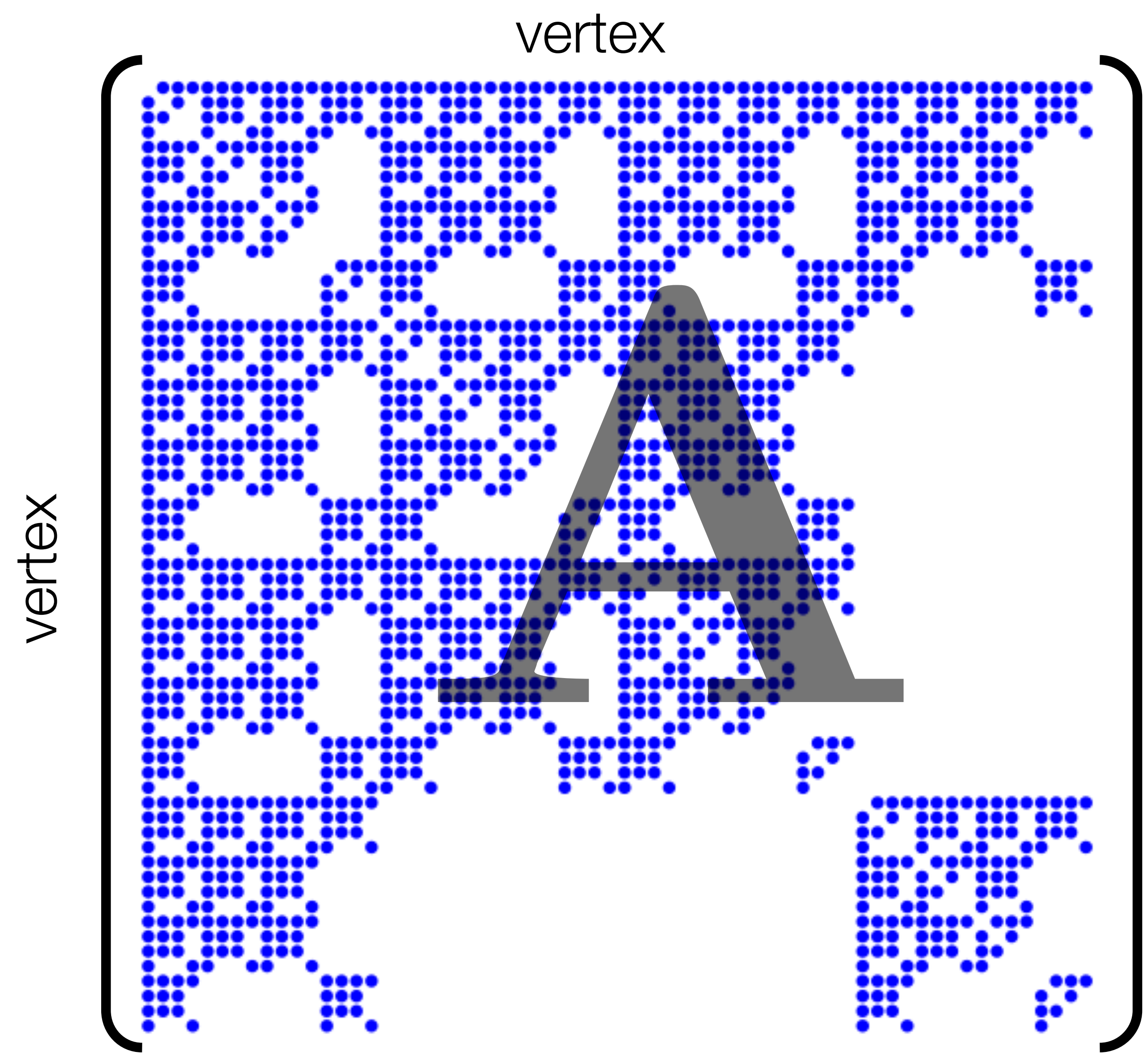
4M nodes, 35M edges



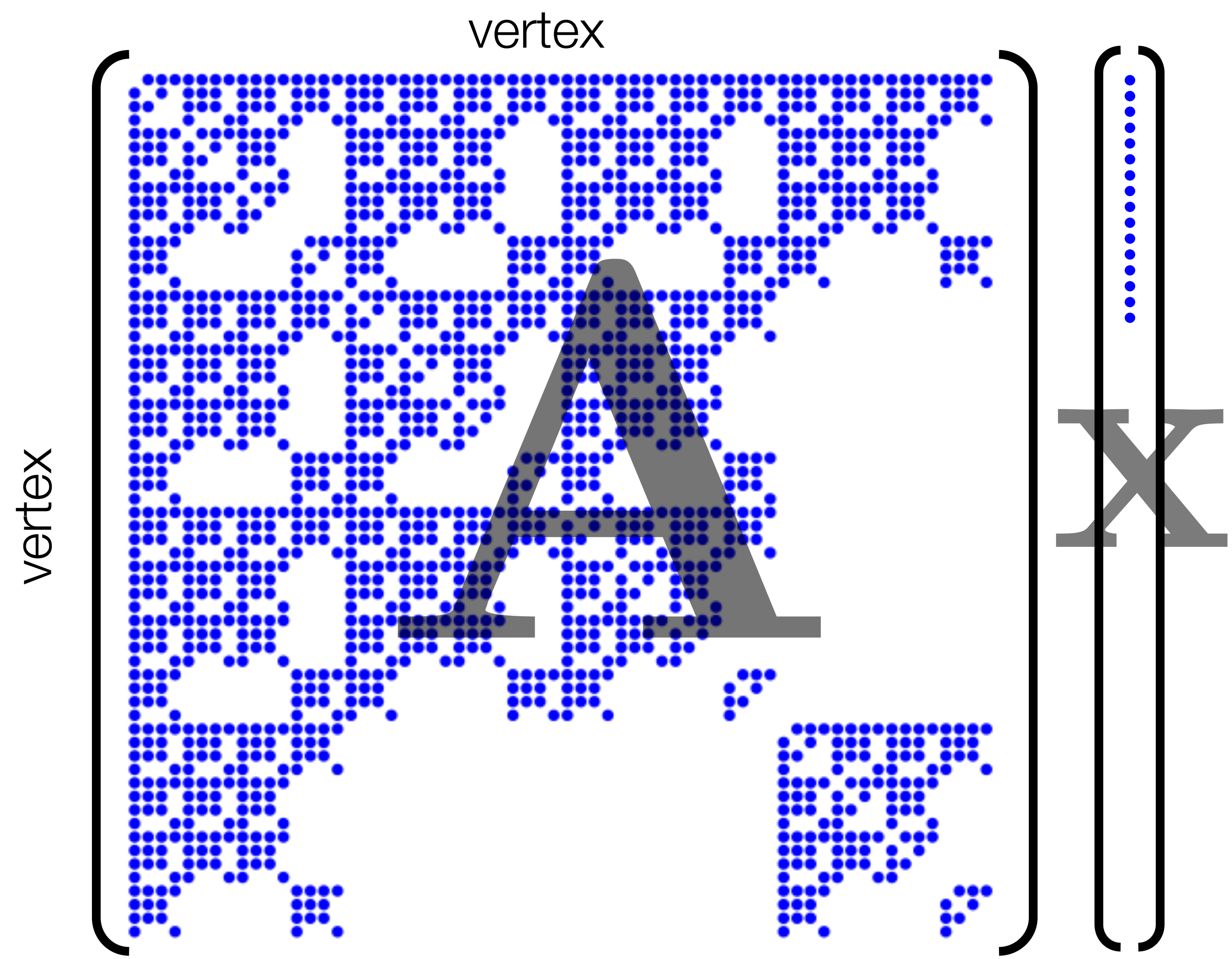
Blue: TPower JMLR’13 Green: GreedyFeige Algorithmica ’01 Yellow: GreedyRavi OR’94

How we do it

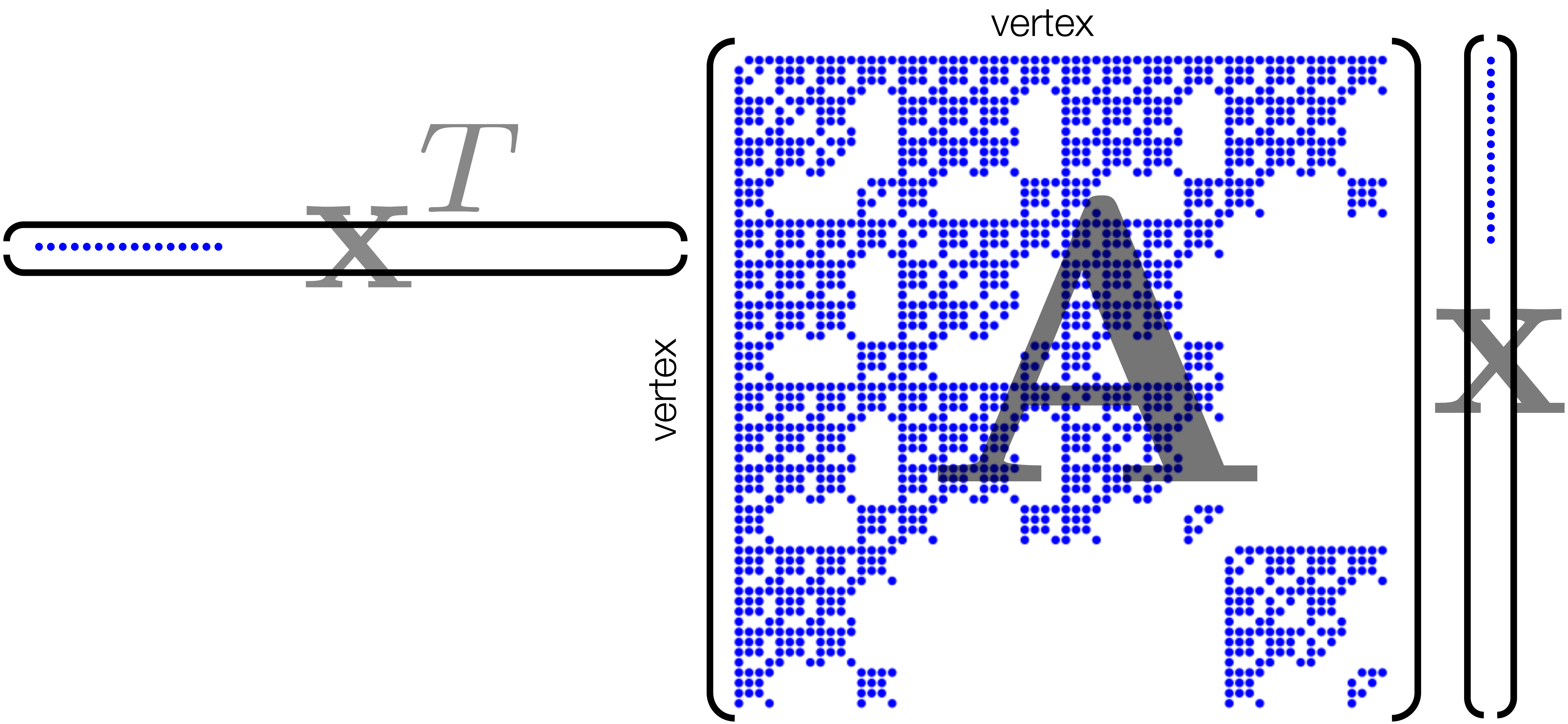
DkS via Quadratic Optimization



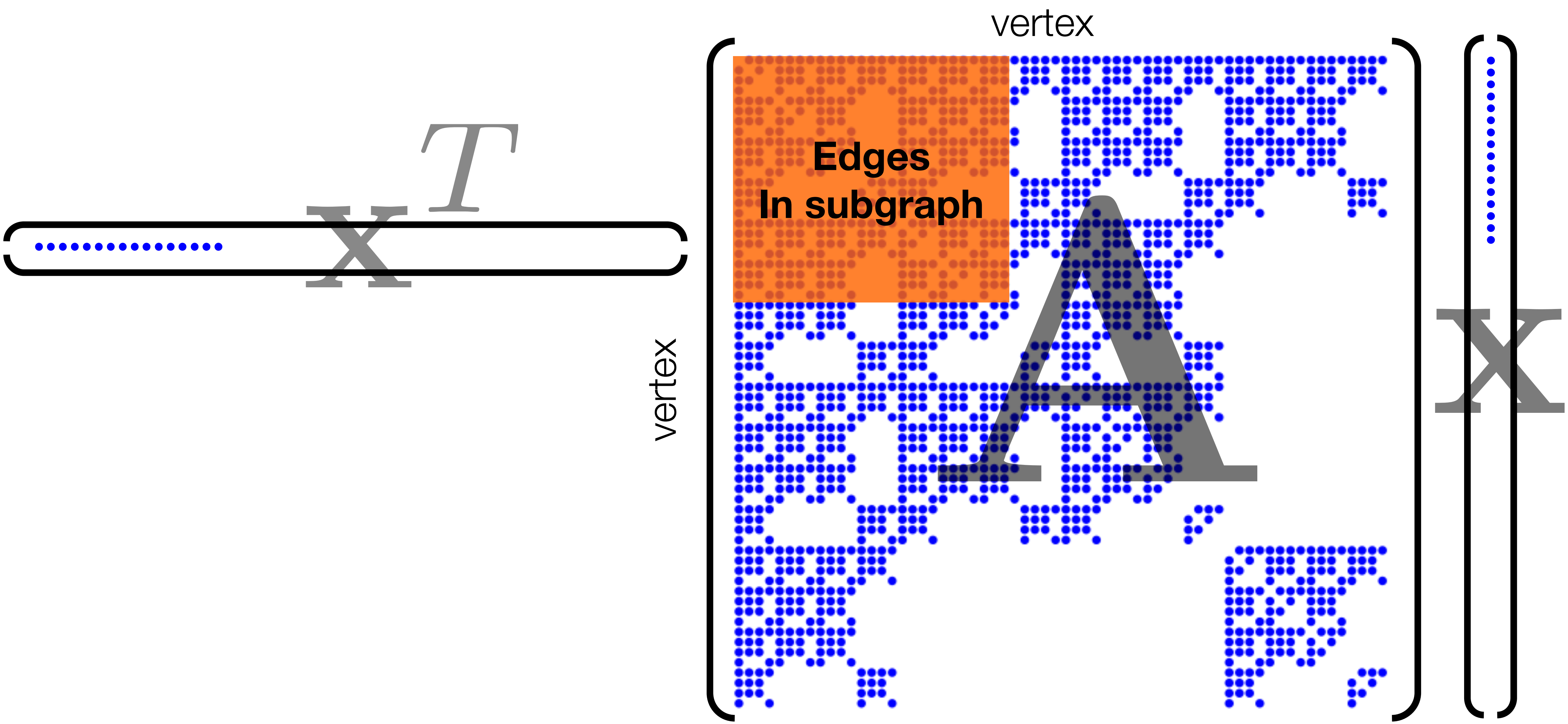
DkS via Quadratic Optimization



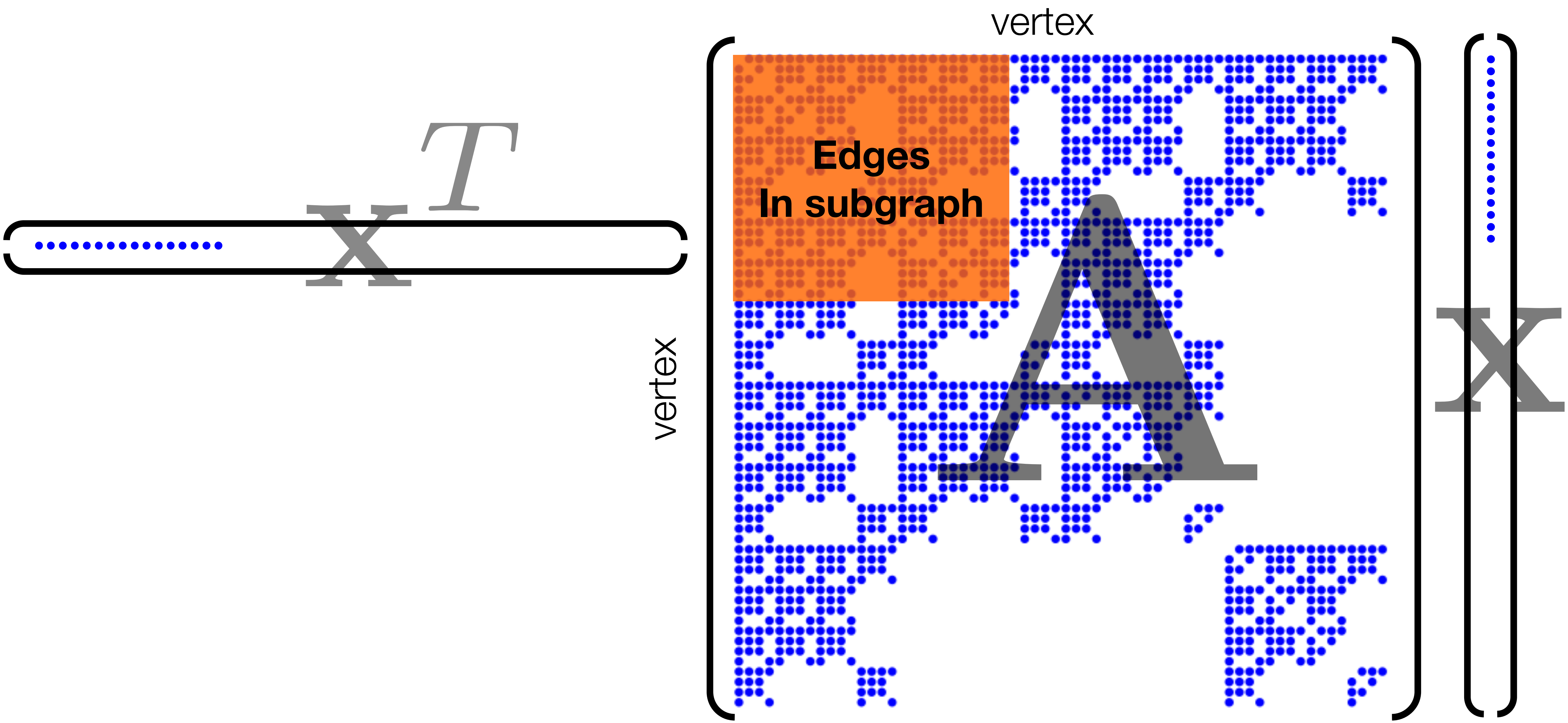
DkS via Quadratic Optimization



DkS via Quadratic Optimization



DkS via Quadratic Optimization



DkS: $$\mathrm{OPT} = \max_{\substack{\mathbf{x} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

DkS via Bilinear Optimization

DkS:

$$\mathrm{OPT} = \max_{\substack{\mathbf{x} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

DkS via Bilinear Optimization

DBkS:

$$\mathrm{OPT} = \max_{\substack{\mathbf{x},\mathbf{y} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = \|\mathbf{y}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{y}$$

DkS:

$$\mathrm{OPT} = \max_{\substack{\mathbf{x} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

DkS via Bilinear Optimization

DBkS:
$$\mathrm{OPT} = \max_{\substack{\mathbf{x},\mathbf{y}\in\{0,1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0=\|\mathbf{y}\|_0=k}} \mathbf{x}^T\mathbf{A}\mathbf{y}$$

DkS:
$$\mathrm{OPT} = \max_{\substack{\mathbf{x}\in\{0,1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0=k}} \mathbf{x}^T\mathbf{A}\mathbf{x}$$

DkS via Bilinear Optimization

DBkS:

$$\mathrm{OPT} = \max_{\substack{\mathbf{x},\mathbf{y} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = \|\mathbf{y}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{y}$$

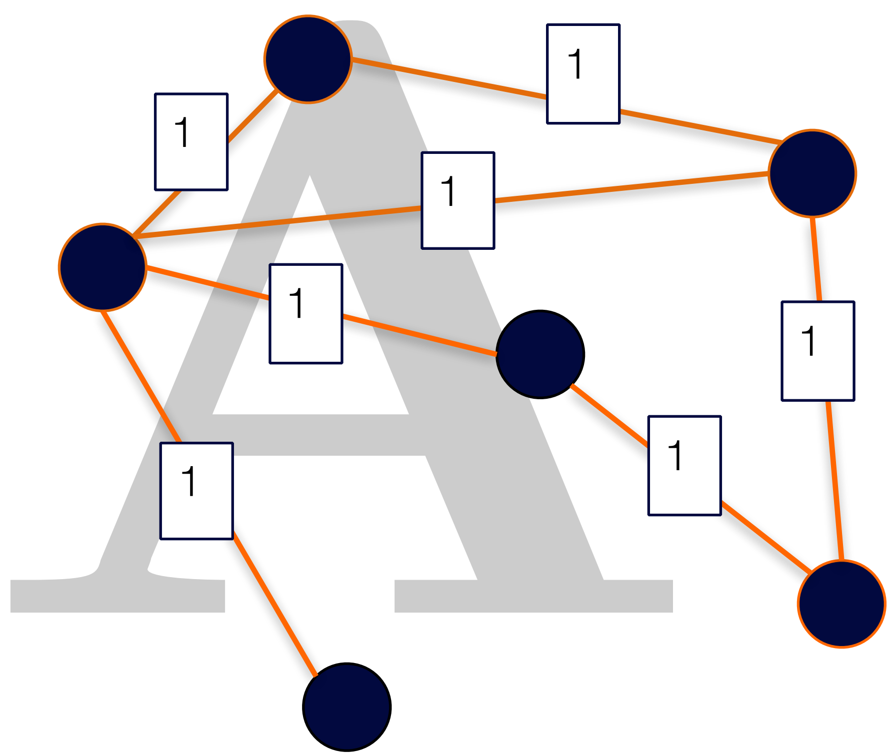
Lemma:
ρ-approximation for DBkS = ½ρ-approximation for DkS

DkS:

$$\mathrm{OPT} = \max_{\substack{\mathbf{x} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

DkS via Bilinear Optimization

DBkS: $$\mathrm{OPT} = \max_{\substack{\mathbf{x},\mathbf{y} \in \{0, 1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = \|\mathbf{y}\|_0 = k}} \mathbf{x}^T \mathbf{A} \mathbf{y}$$



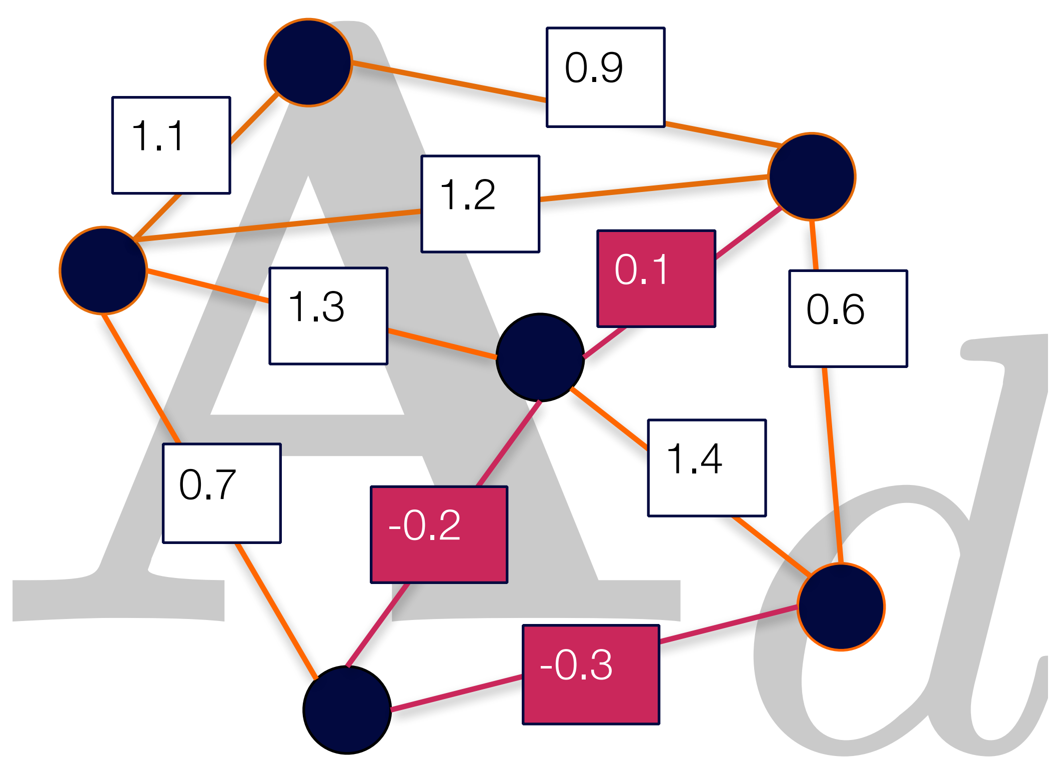
Low-Rank Approximation

DBkS:

$$\mathrm{OPT}_d = \max_{\substack{\mathbf{x},\mathbf{y} \in \{0,1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0 = \|\mathbf{y}\|_0 = k}} \mathbf{x}^T \mathbf{A}_d \mathbf{y}$$

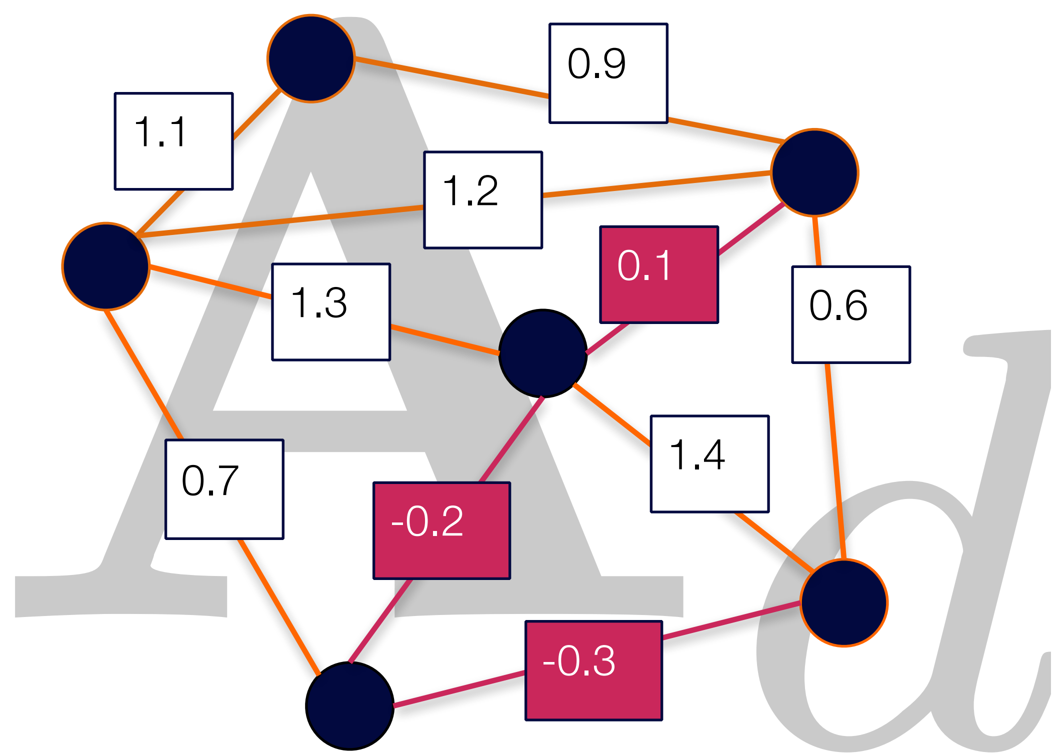
Low-Rank Approximation

DBkS: $$\mathrm{OPT}_d = \max_{\substack{\mathbf{x},\mathbf{y}\in\{0,1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0=\|\mathbf{y}\|_0=k}} \mathbf{x}^T \mathbf{A}_d \mathbf{y}$$



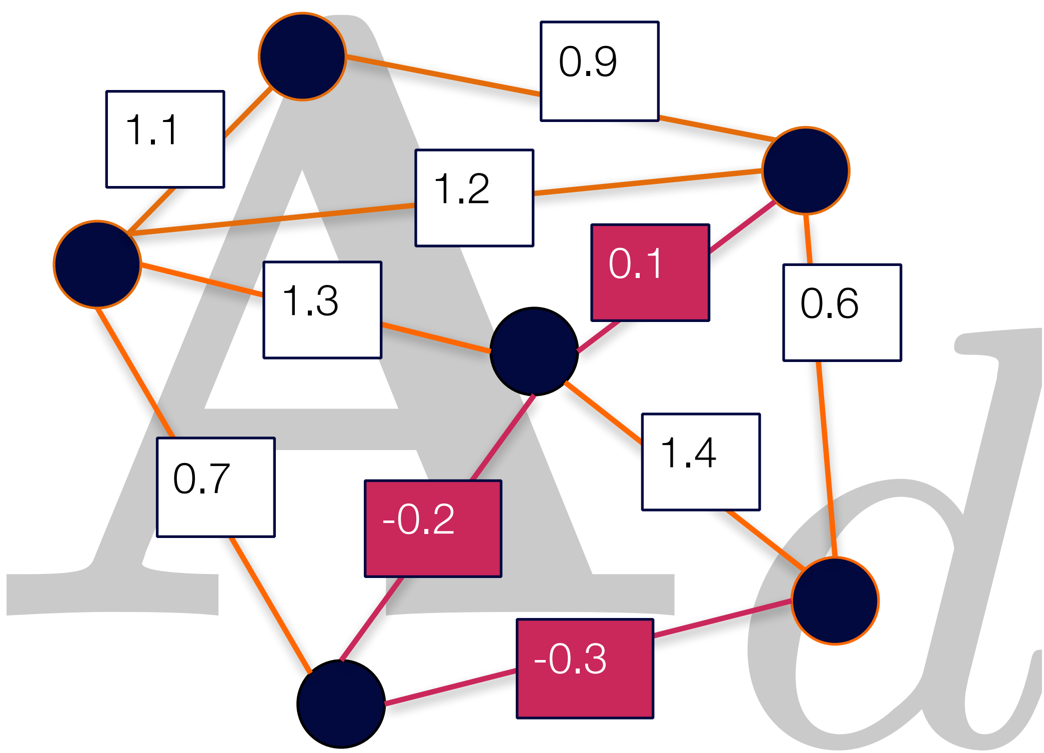
Low-Rank Approximation

DBkS:
$$\mathrm{OPT}_d = \max_{\substack{\mathbf{x},\mathbf{y}\in\{0,1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0=\|\mathbf{y}\|_0=k}} \mathbf{x}^T \mathbf{A}_d \mathbf{y}$$



Low-Rank Approximation

DBkS: $$\mathrm{OPT}_d = \max_{\substack{\mathbf{x},\mathbf{y}\in\{0,1/\sqrt{k}\}^n \\ \|\mathbf{x}\|_0=\|\mathbf{y}\|_0=k}} \mathbf{x}^T \mathbf{A}_d \mathbf{y}$$



Efficiently solvable

How the Low-Rank Solver Works

Naïvely: Check all $\binom{n}{k}$ subgraphs

Rank-1 case:

$$\mathbf{A}_1 = \mathbf{v}\mathbf{v}^T$$

Q: Maximize the product of two numbers

$$\max_{\substack{\mathbf{x},\mathbf{y} \in \{0,1\}^n \\ \|\mathbf{x}\|_0 = \|\mathbf{y}\|_0 = k}} (\mathbf{x}^T \mathbf{v}) \cdot (\mathbf{v}^T \mathbf{y})$$

A: Maximize each number **individually**

How the Rank-1 Solver Works

$$\max_{\substack{\mathbf{x},\mathbf{y}\in\{0,1\}^n \\ \|\mathbf{x}\|_0=\|\mathbf{y}\|_0=k}} (\mathbf{x}^T \quad) \cdot (\quad ^T \mathbf{y})$$

top-k set: the k-largest coordinates of a vector,
e.g., if k =2, then top-2 set = {3,4}

Intuition: *x*, *y* pick the top-k set of *v*.

How the Rank-2 Solver Works

$$\max\left(\mathbf{x}^T \begin{bmatrix} 1 & 5 \\ 2 & 2 \\ 3 & 7 \\ 4 & 0 \end{bmatrix}\right)\left(\begin{bmatrix} 1 & 5 \\ 2 & 2 \\ 3 & 7 \\ 4 & 0 \end{bmatrix}\right)^T \mathbf{y}$$

Intuition: *x*, *y* pick the top-*k* set of a vector from a 2-dimensional span.

Q: How many top-k sets are there in a 2-dimensional span?

Based on **Spannogram** [Asteris, Papail., Karystinos, ISIT2011]

Theorem: # top-*k* sets in a d-dimensional span: $\binom{d}{\frac{d}{2}}\binom{n}{d} = O(n^d)$

Spannogram: Traverses all of them efficiently

How the Rank-2 Solver Works

$$\max\left(\mathbf{x}^T \begin{bmatrix} 1 & 5 \\ 2 & 2 \\ 3 & 7 \\ 4 & 0 \end{bmatrix}\right)\left(\begin{bmatrix} 1 & 5 \\ 2 & 2 \\ 3 & 7 \\ 4 & 0 \end{bmatrix}^T \mathbf{y}\right)$$

Intuition: *x*, *y* pick the top-*k* set of a vector from a 2-dimensional span.

Randomized algorithm

Take random points: $s_1, \ldots, s_{1/\epsilon^d} \in \mathrm{span}(v_1, \ldots, v_d)$

How the Rank-2 Solver Works

$$\max(\mathbf{x}^T \begin{bmatrix} 1 & 5 \\ 2 & 2 \\ 3 & 7 \\ 4 & 0 \end{bmatrix})(\begin{bmatrix} 1 & 5 \\ 2 & 2 \\ 3 & 7 \\ 4 & 0 \end{bmatrix}^T \mathbf{y}$$

Intuition: *x*, *y* pick the top-*k* set of a vector from a 2-dimensional span.

Randomized algorithm

Take random points: $s_1, \ldots, s_{1/\epsilon^d} \in \mathrm{span}(v_1, \ldots, v_d)$

Practically linear time

Implementation

MapReduce Implementation

```
def spannogram_mapper(self, coordinate, values):
    i = coordinate
    for j in range(int(self.options.rowcount)):
        yield j, values

def spannogram_reducer(self, coordinate, values):
    k = 5
    V = list(values)
    i = coordinate
    Vc = []
    opt_support = [];
    opt_metric = 0;
    for j in range(i+1, int(self.options.rowcount)):
        x = []
        Vc = []
        Vtemp = []
        # compute c_ij intersection vector
        x = [V[i][l]-V[j][l] for l in range(2)]
        # cumpute v_ij = Vc_ij
        Vc = [V[l][0]*x[1]-V[l][1]*x[0] for l in range(int(self.options.rowcount))]
        # find top and bottom support
        top_support_pos = zip(*heapq.nlargest(k, enumerate(Vc), key=operator.itemgetter(1)))[0]
        top_support_neg = zip(*heapq.nsmallest(k, enumerate(Vc), key=operator.itemgetter(1)))[0]
        # compute support metric
        Vtemp = [V[s] for s in top_support_pos]
        metric_pos = sum([x**2 for x in [sum(a) for a in zip(*Vtemp)]])
        Vtemp = []
        Vtemp = [V[s] for s in top_support_neg]
        metric_neg = sum([x**2 for x in [sum(a) for a in zip(*Vtemp)]])
        # find locally optimal support
        metric_list = [opt_metric, metric_pos, metric_neg]
        metric_index = metric_list.index(max(metric_list))
        opt_support = [opt_support, top_support_pos, top_support_neg][metric_index]
        opt_metric = max(metric_list)

        yield i, [opt_metric, opt_support]
```

MapReduce Implementation

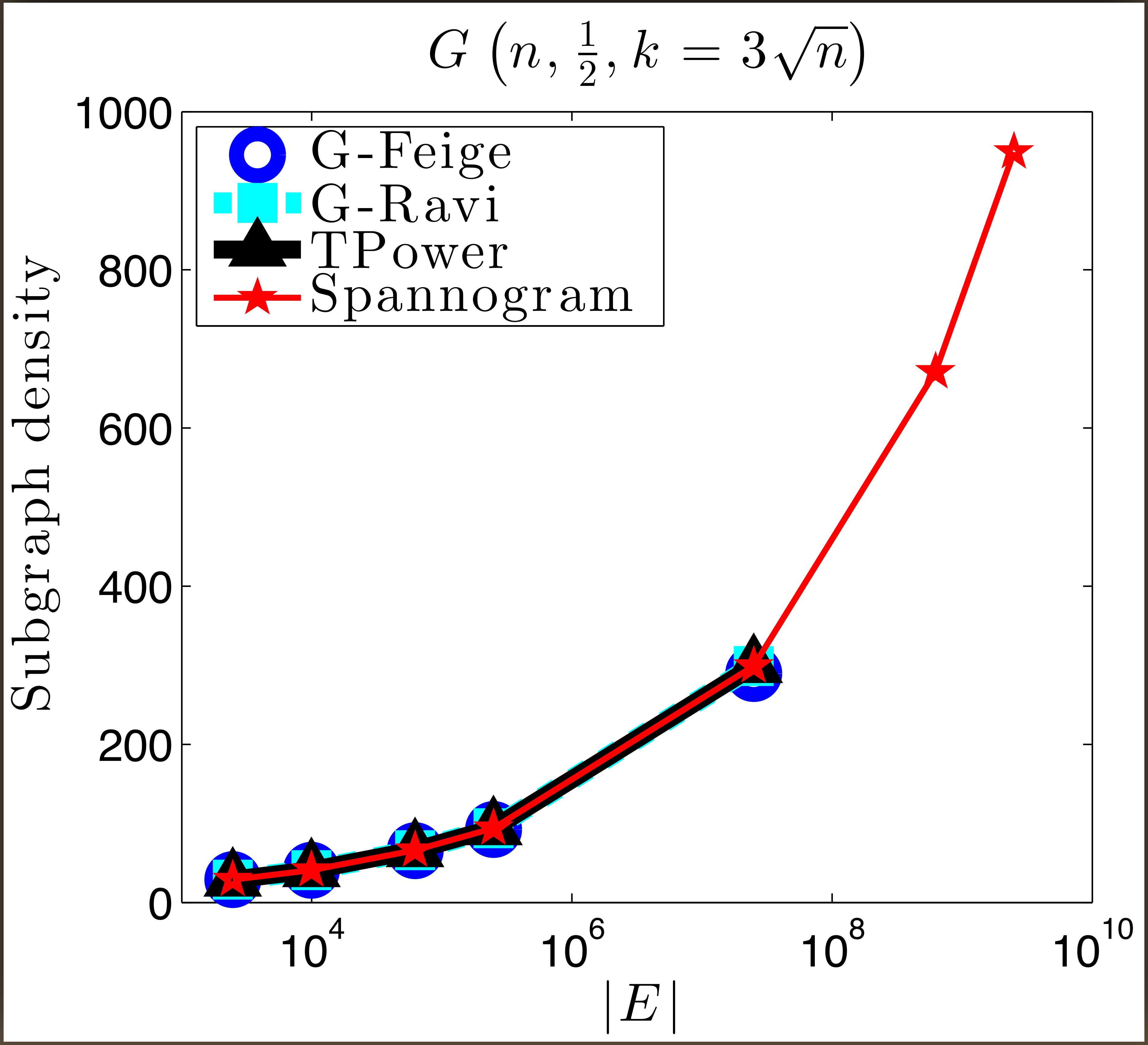
```python
def spannogram_mapper(self, coordinate, values):
    i = coordinate
    for j in range(int(self.options.rowcount)):
        yield j, values

def spannogram_reducer(self, coordinate, values):
    k = 5
    V = list(values)
    i = coordinate
    Vc = []
    opt_support = [];
    opt_metric = 0;
    for j in range(i+1, int(self.options.rowcount)):
        x = []
        Vc = []
        Vtemp = []
        # compute c_ij intersection vector
        x = [V[i][l]-V[j][l] for l in range(2)]
        # cumpute v_ij = Vc_ij
        Vc = [V[l][0]*x[1]-V[l][1]*x[0] for l in range(int(self.options.rowcount))]
        # find top and bottom support
        top_support_pos = zip(*heapq.nlargest(k, enumerate(Vc), key=operator.itemgetter(1)))[0]
        top_support_neg = zip(*heapq.nsmallest(k, enumerate(Vc), key=operator.itemgetter(1)))[0]
        # compute support metric
        Vtemp = [V[s] for s in top_support_pos]
        metric_pos = sum([x**2 for x in [sum(a) for a in zip(*Vtemp)]])
        Vtemp = []
        Vtemp = [V[s] for s in top_support_neg]
        metric_neg = sum([x**2 for x in [sum(a) for a in zip(*Vtemp)]])
        # find locally optimal support
        metric_list = [opt_metric, metric_pos, metric_neg]
        metric_index = metric_list.index(max(metric_list))
        opt_support = [opt_support, top_support_pos, top_support_neg][metric_index]
        opt_metric = max(metric_list)

        yield i, [opt_metric, opt_support]
```

git.io/spannogram

Billion-scale Graphs



Conclusions

Conclusions

- New combinatorial approx. algorithm for DkS.

Conclusions

- New combinatorial approx. algorithm for DkS.
- Graph-dependent spectral bounds:
  OPT within 70% in most experiments.

Conclusions

- New combinatorial approx. algorithm for DkS.
- Graph-dependent spectral bounds:
  OPT within 70% in most experiments.
- Bound could be trivial in the worst case.

Conclusions

- New combinatorial approx. algorithm for DkS.
- Graph-dependent spectral bounds:
  OPT within 70% in most experiments.
- Bound could be trivial in the worst case.
- Empirically outperforms previous state of the art

Conclusions

- New combinatorial approx. algorithm for DkS.
- Graph-dependent spectral bounds:
  OPT within 70% in most experiments.
- Bound could be trivial in the worst case.
- Empirically outperforms previous state of the art

Conclusions

- New combinatorial approx. algorithm for DkS.
- Graph-dependent spectral bounds:
  OPT within 70% in most experiments.
- Bound could be trivial in the worst case.
- Empirically outperforms previous state of the art
- Highly scalable implementation

Thank you

References

Amazon Web Services, Elastic Map Reduce. URL http://aws.amazon.com/elasticmapreduce/.

MRJob. URL http://pythonhosted.org/mrjob/.

Ames, Brendan PW. *Convex relaxation for the planted clique, biclique, and clustering problems.* PhD thesis, University of Waterloo, 2011.

Arora, Sanjeev, Karger, David, and Karpinski, Marek. Polynomial time approximation schemes for dense instances of np-hard problems. In *STOC*, 1995.

Asahiro, Yuichi, Iwama, Kazuo, Tamaki, Hisao, and Tokuyama, Takeshi. Greedily finding a dense subgraph. *Journal of Algorithms*, 34(2):203–221, 2000.

Asteris, Megasthenis, Papailiopoulos, Dimitris S, and Karystinos, George N. Sparse principal component of a rank-deficient matrix. In *IEEE ISIT 2011*.

Bahmani, Bahman, Kumar, Ravi, and Vassilvitskii, Sergei. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

Bhaskara, Aditya, Charikar, Moses, Chlamtac, Eden, Feige, Uriel, and Vijayaraghavan, Aravindan. Detecting high log-densities: an $\mathsf{O}(n^{1/4})$ approximation for densest k-subgraph. In *STOC*, 2010.

Boutsidis, Christos, Mahoney, Michael W, and Drineas, Petros. An improved approximation algorithm for the column subset selection problem. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 968–977. Society for Industrial and Applied Mathematics, 2009.

Cormen, Thomas H, Leiserson, Charles E, Rivest, Ronald L, and Stein, Clifford. *Introduction to algorithms.* MIT press, 2001.

d'Aspremont, Alexandre et al. Weak recovery conditions using graph partitioning bounds. 2010.

Dourisboure, Yon, Geraci, Filippo, and Pellegrini, Marco. Extraction and classification of dense communities in the web. In *WWW*, 2007.

Feige, Uriel and Langberg, Michael. Approximation algorithms for maximization problems arising in graph partitioning. *Journal of Algorithms*, 41(2):174–211, 2001.

Feige, Uriel, Peleg, David, and Kortsarz, Guy. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

Gibson, David, Kumar, Ravi, and Tomkins, Andrew. Discovering large dense subgraphs in massive graphs. In *PVLDB*, 2005.

Gittens, Alex, Kambadur, Prabhanjan, and Boutsidis, Christos. Approximate spectral clustering via randomized sketching. *arXiv preprint arXiv:1311.2854*, 2013.

Halko, Nathan, Martinsson, Per-Gunnar, and Tropp, Joel A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

Hu, Haiyan, Yan, Xifeng, Huang, Yu, Han, Jiawei, and Zhou, Xianghong Jasmine. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl 1):i213–i221, 2005.

Jethava, Vinay, Martinsson, Anders, Bhattacharyya, Chiranjib, and Dubhashi, Devdatt. The lovasz theta function, svms and finding large dense subgraphs. In *NIPS*, 2012.

Karystinos, George N and Liavas, Athanasios P. Efficient computation of the binary vector that maximizes a rank-deficient quadratic form. *IEEE Trans. IT*, 56(7):3581–3593, 2010.

Khot, Subhash. Ruling out ptas for graph min-bisection, densest subgraph and bipartite clique. In *FOCS*, 2004.

Lin, Jimmy and Schatz, Michael. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pp. 78–85. ACM, 2010.

Mahoney, Michael W and Drineas, Petros. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702, 2009.

Meng, Xiangrui and Mahoney, Michael W. Robust regression on mapreduce. *ICML 2013, (to appear)*.

Miller, B, Bliss, N, and Wolfe, P. Subgraph detection using eigenvector l1 norms. In *NIPS*, 2010.

Papailiopoulos, Dimitris S, Dimakis, Alexandros G, and Korokythakis, Stavros. Sparse pca through low-rank approximations. *arXiv preprint arXiv:1303.0551*, 2013.

Ravi, Sekharipuram S, Rosenkrantz, Daniel J, and Tayi, Giri K. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

Rokhlin, Vladimir, Szlam, Arthur, and Tygert, Mark. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31 (3):1100–1124, 2009.

Saha, Barna, Hoch, Allison, Khuller, Samir, Raschid, Louiqa, and Zhang, Xiao-Ning. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Research in Computational Molecular Biology*, pp. 456–472. Springer, 2010.

Srivastav, Anand and Wolf, Katja. *Finding dense subgraphs with semidefinite programming.* Springer, 1998.

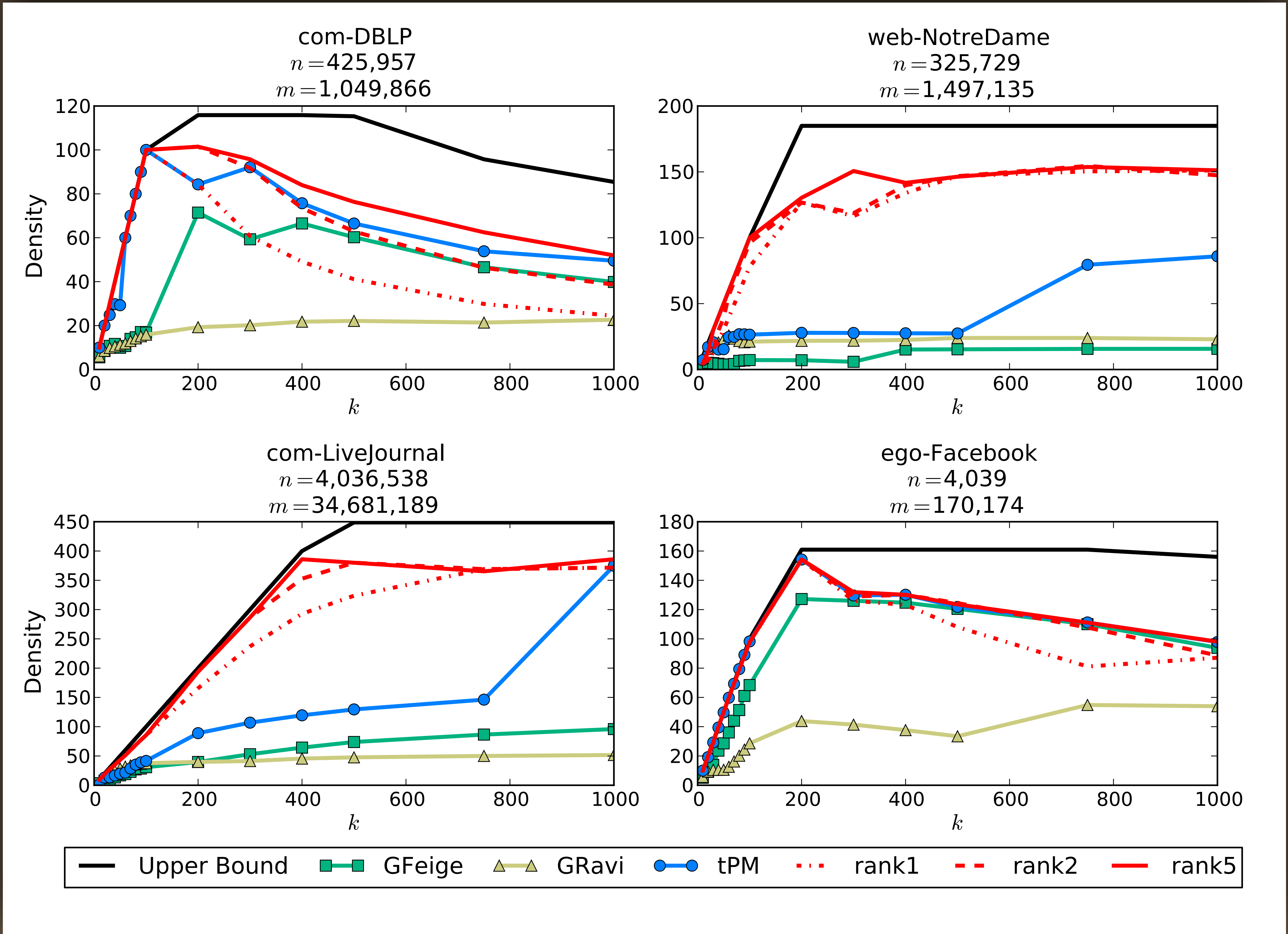
Suzuki, Akiko and Tokuyama, Takeshi. Dense subgraph problems with output-density conditions. In *Algorithms and Computation*, pp. 266–276. Springer, 2005.

Wyner, Aaron D. Random packings and coverings of the unit n-sphere. *Bell System Technical Journal*, 46(9): 2111–2118, 1967.

Yuan, Xiao-Tong and Zhang, Tong. Truncated power method for sparse eigenvalue problems. *arXiv preprint arXiv:1112.2679*, 2011.

Backup slides

Other experiments



Randomized Algorithm

Step 1

Take random points: $s_1, \ldots, s_{1/\epsilon^d} \in \mathrm{span}(v_1, \ldots, v_d)$

Step 2

Find largest k entries: $\mathrm{top}_k(\mathbf{s}_i)$

Step 3

Compute density of corresponding subgraph

Randomized Algorithm

Step 1

Take random points: $s_1, \ldots, s_{1/\epsilon^d} \in \text{span}(v_1, \ldots, v_d)$

Step 2

Find largest k entries: $\text{top}_k(\mathbf{s}_i)$

Step 3

Compute density of corresponding subgraph

Practically **linear time**