

Heat Kernel Based Community Detection



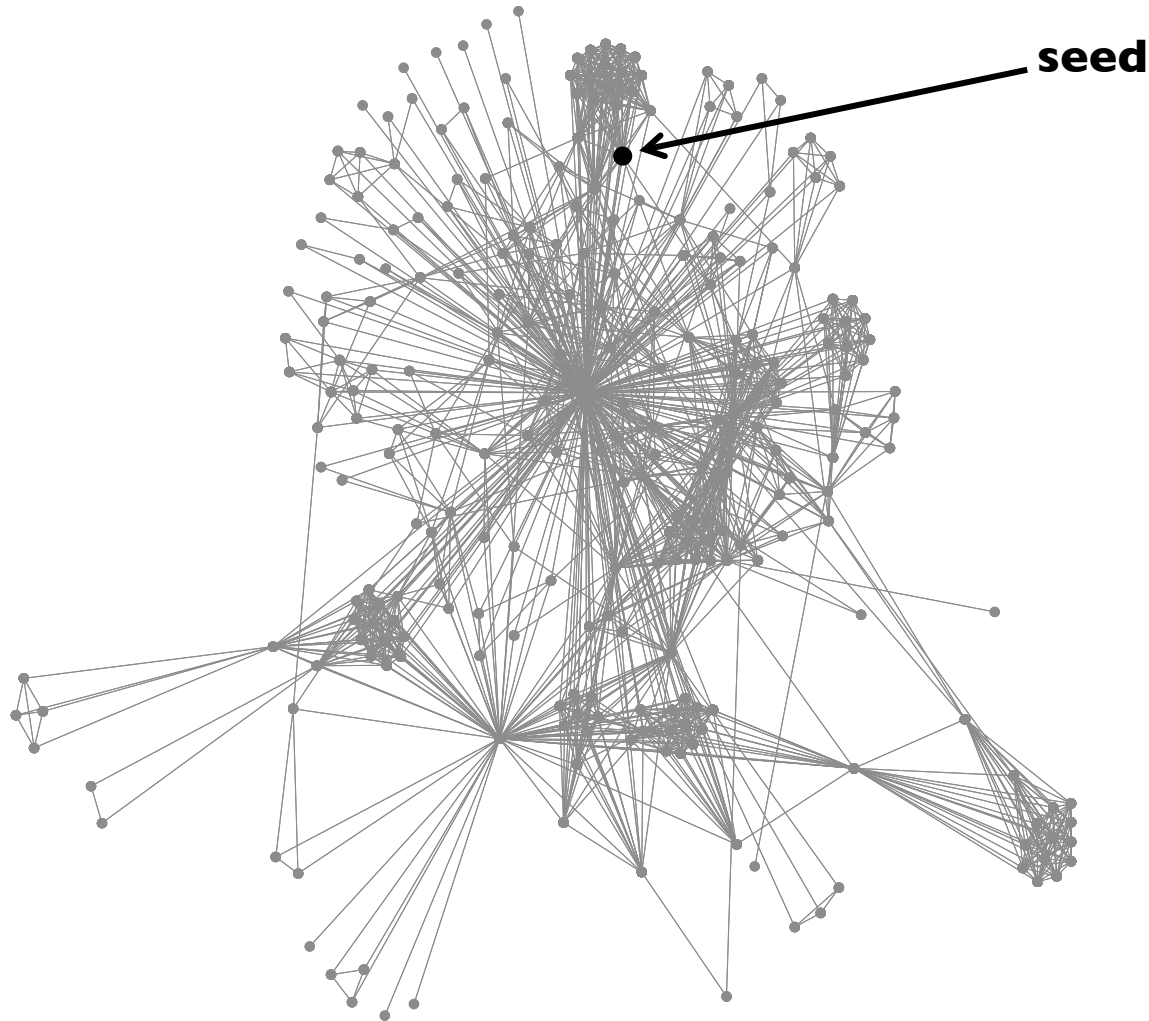
Joint with
David F. Gleich,
(Purdue), supported by
NSF CAREER
1149756-CCF

PURDUE
UNIVERSITY

Kyle Kloster
Purdue University

Local Community Detection

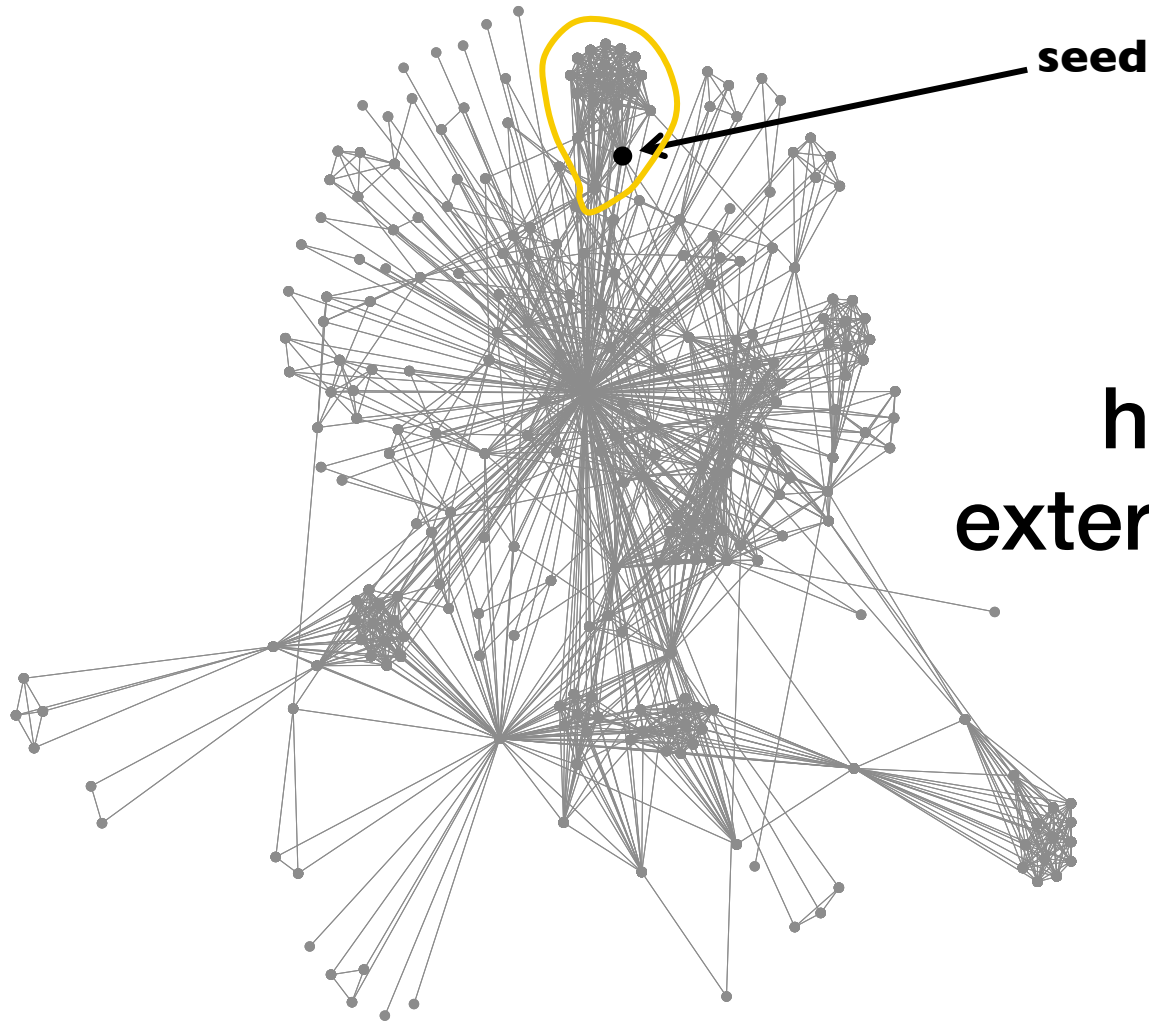
Given seed(s) S in G , find a community that contains S .



“Community” ?

Local Community Detection

Given seed(s) S in G , find a community that contains S .



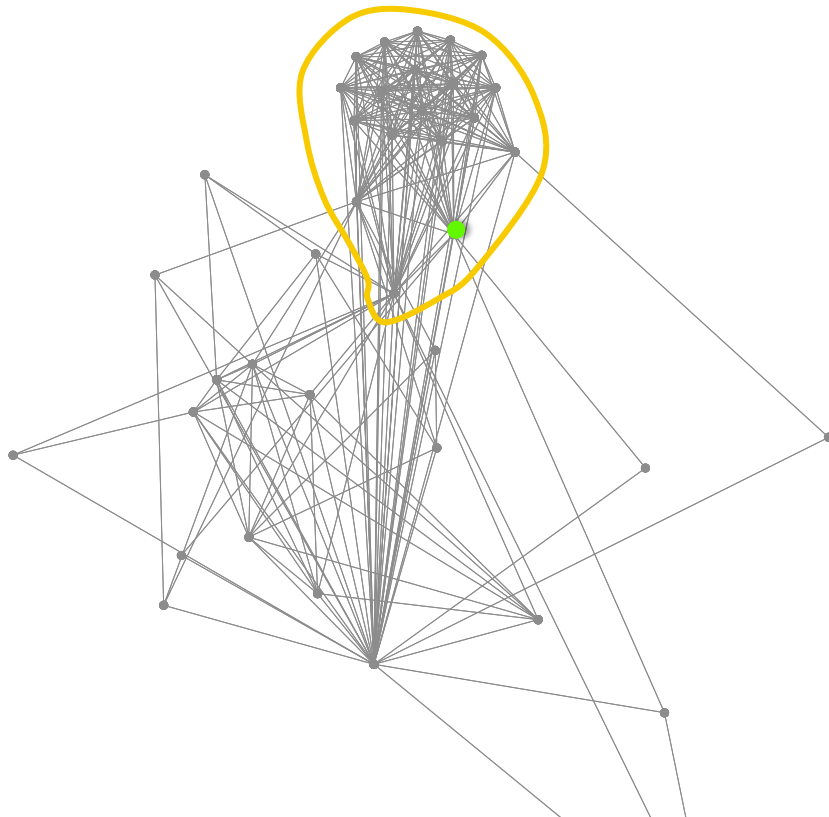
“Community” ?

**high internal, low
external connectivity**

Low-conductance sets are communities

$$\text{conductance}(T) = \frac{\# \text{ edges leaving } T}{\# \text{ edge endpoints in } T}$$

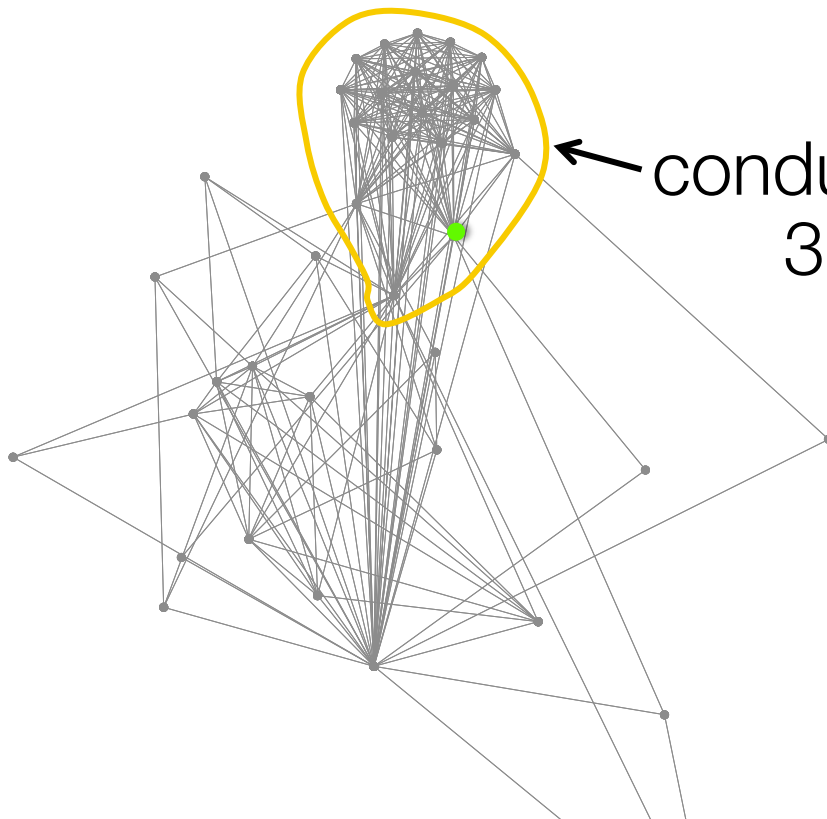
= “chance a random step exits T ”



Low-conductance sets are communities

$$\text{conductance}(T) = \frac{\# \text{ edges leaving } T}{\# \text{ edge endpoints in } T}$$

= “chance a random step exits T ”

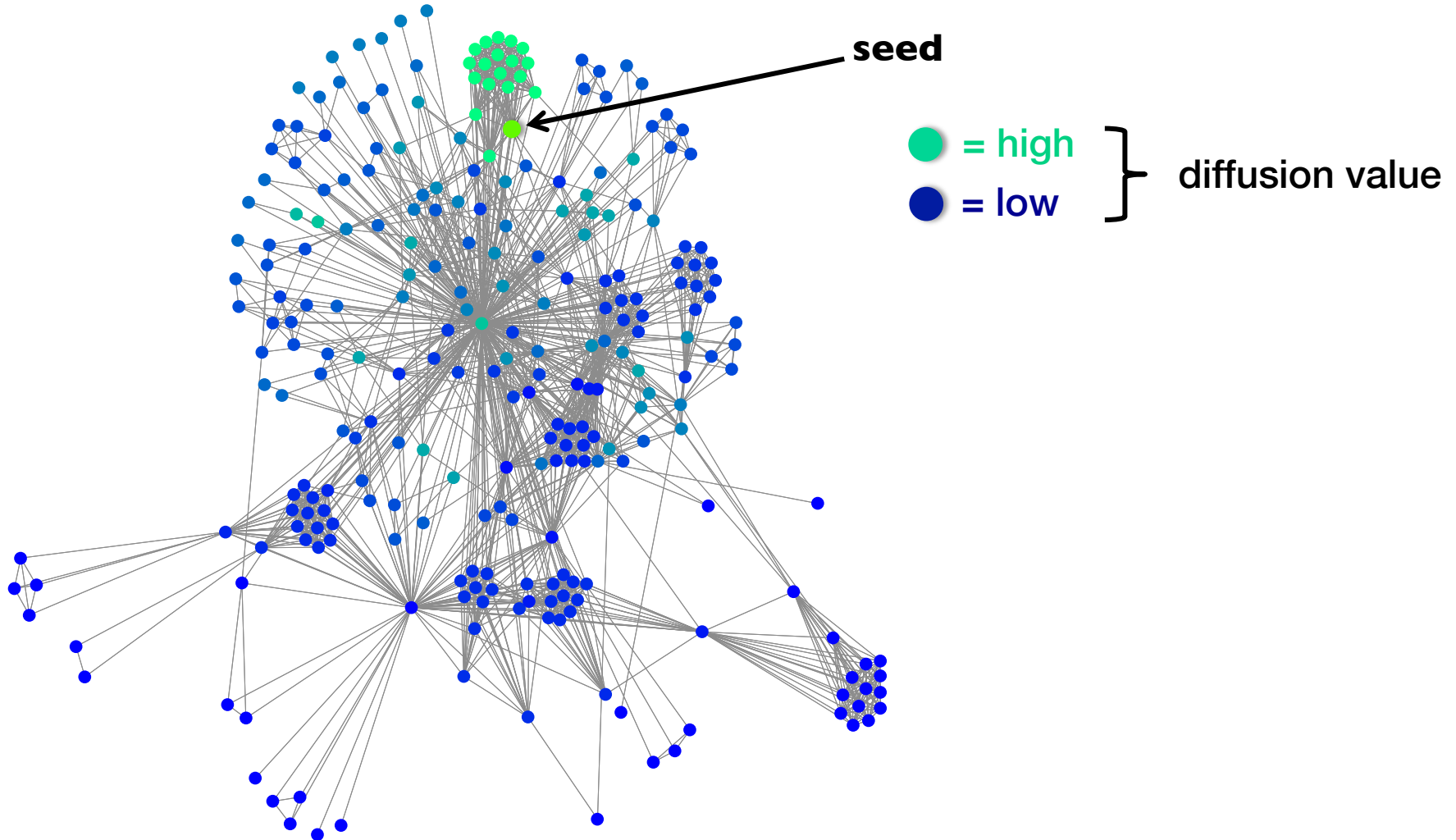


$$\text{conductance}(\text{comm}) = \frac{39}{381} = .102$$

How to find these ?

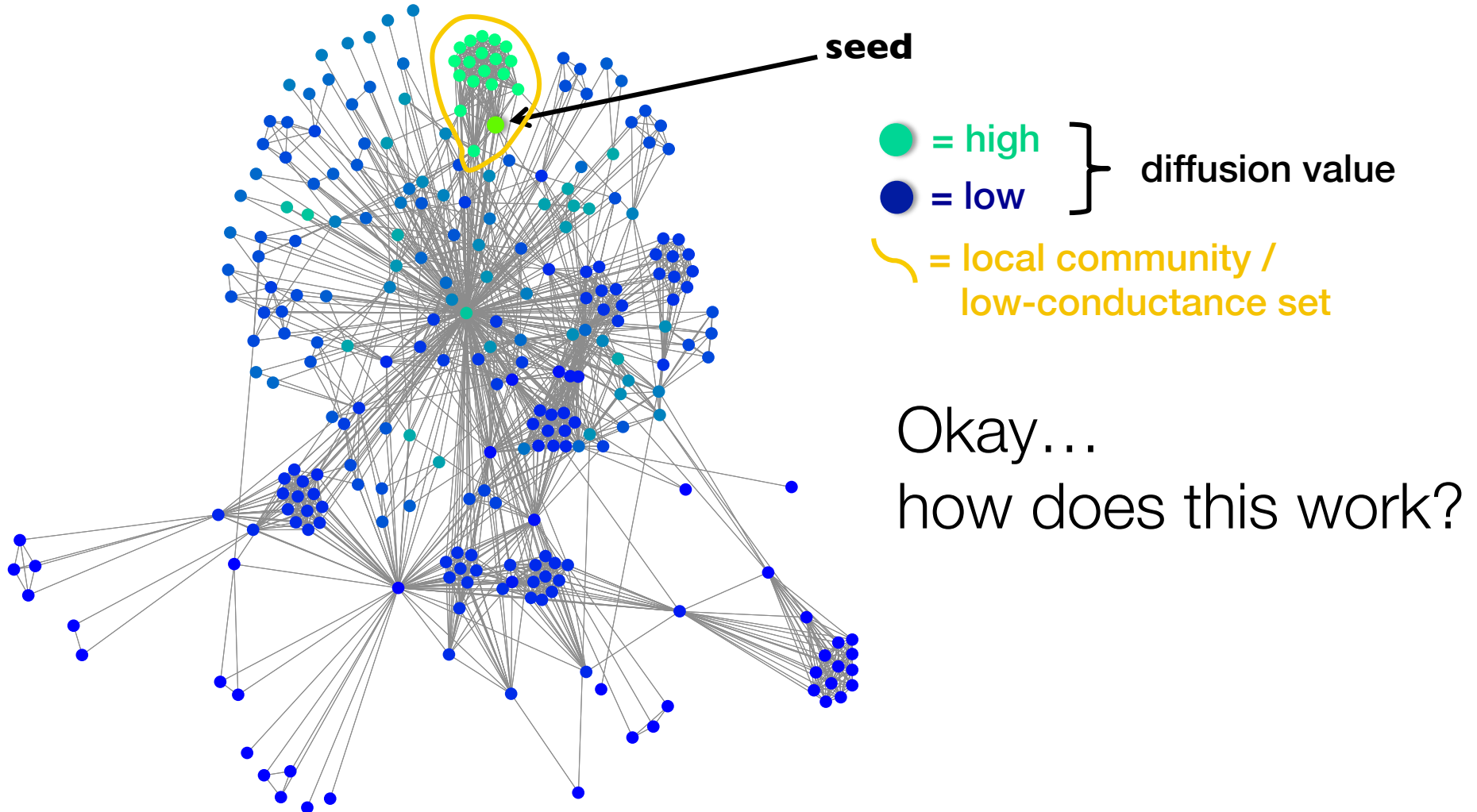
Graph diffusions find low-conductance sets

A diffusion propagates “rank” from a seed across a graph.



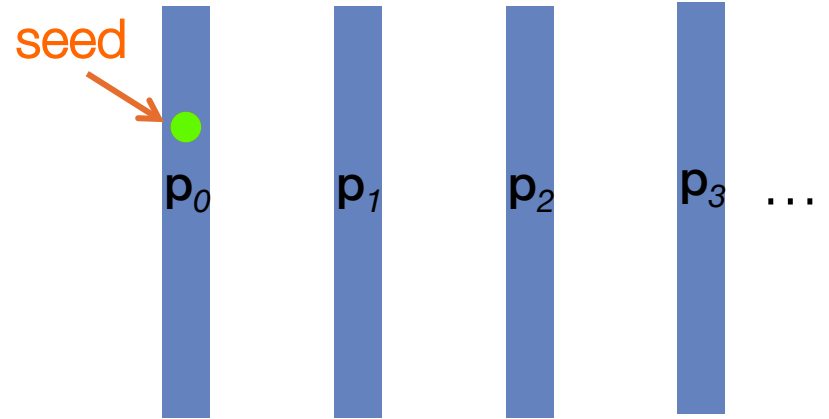
Graph diffusions find low-conductance sets

A diffusion propagates “rank” from a seed across a graph.



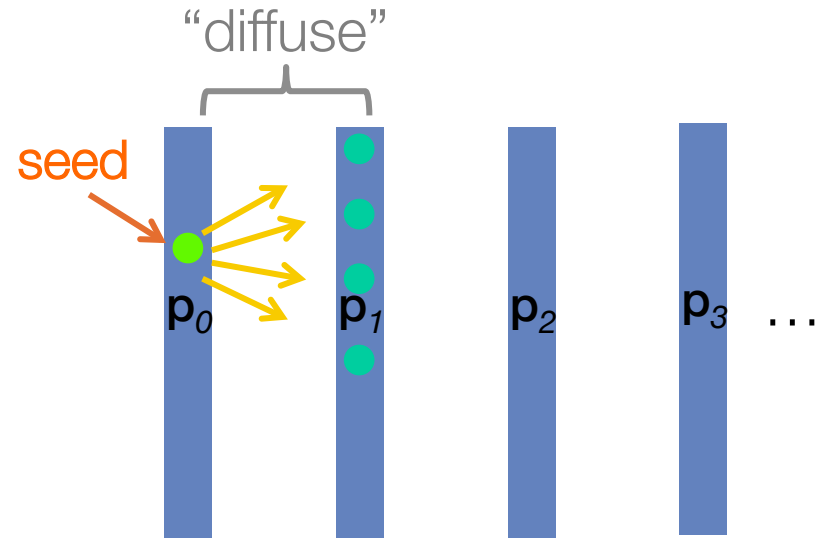
Graph Diffusion

A diffusion models how a mass (green dye, money, popularity) spreads from a seed across a network.



Graph Diffusion

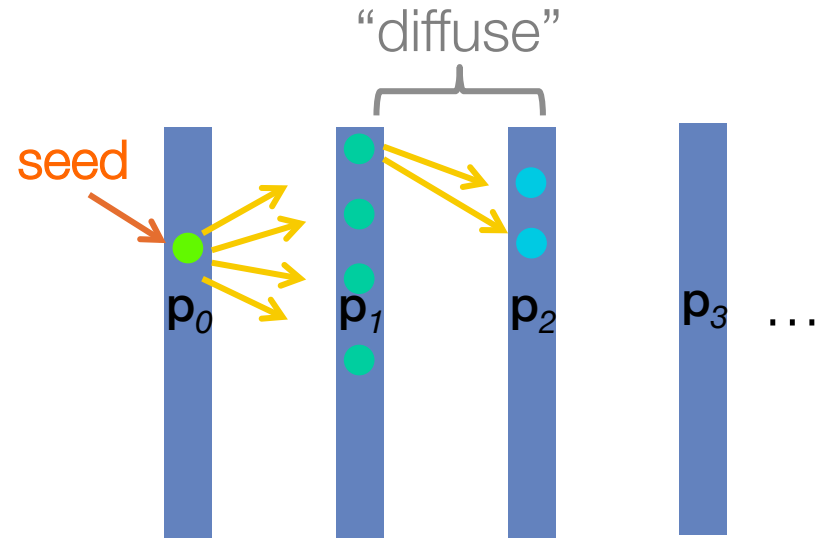
A diffusion models how a mass (green dye, money, popularity) spreads from a seed across a network.



Graph Diffusion

A diffusion models how a mass (green dye, money, popularity) spreads from a seed across a network.

Once mass reaches a node, it propagates to the neighbors, with some decay.

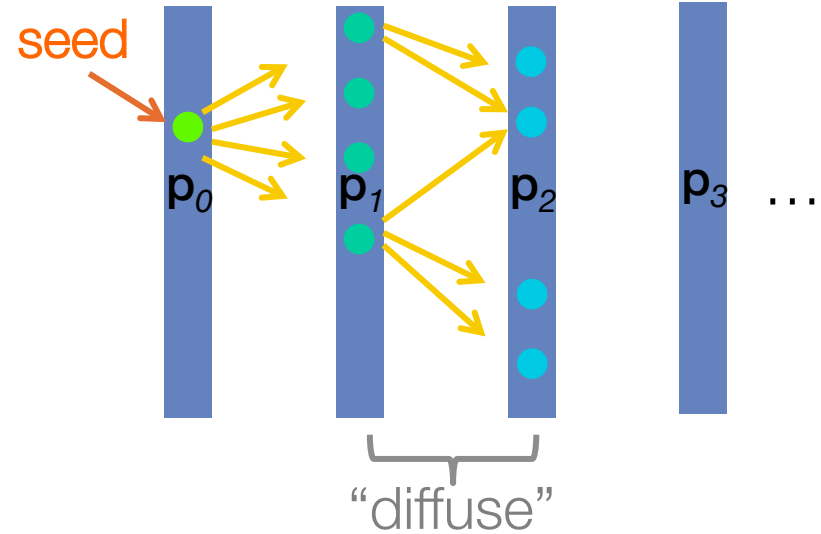


“decay”: dye dilutes, money is taxed, popularity fades

Graph Diffusion

A diffusion models how a mass (green dye, money, popularity) spreads from a seed across a network.

Once mass reaches a node, it propagates to the neighbors, with some decay.



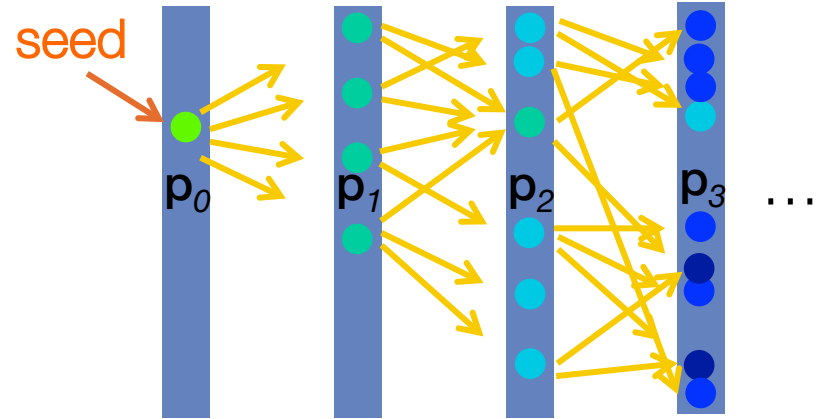
“decay”: dye dilutes, money is taxed, popularity fades

Graph Diffusion

A diffusion models how a mass (green dye, money, popularity) spreads from a seed across a network.

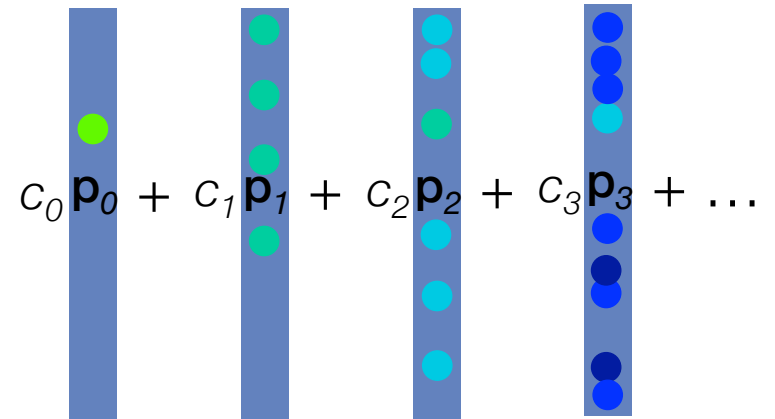
Once mass reaches a node, it propagates to the neighbors, with some decay.

“decay”: dye dilutes, money is taxed, popularity fades



Diffusion score

“diffusion score” of a node =
weighted sum of the mass at that
node during different stages.

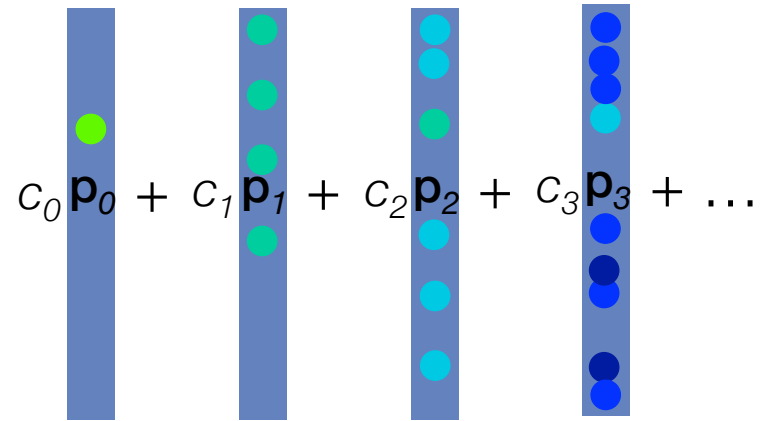


Diffusion score

“diffusion score” of a node =
weighted sum of the mass at that
node during different stages.

diffusion score vector = \mathbf{f}

$$\mathbf{f} = \sum_{k=0}^{\infty} c_k \mathbf{P}^k \mathbf{s}$$



\mathbf{P} = random-walk
transition matrix

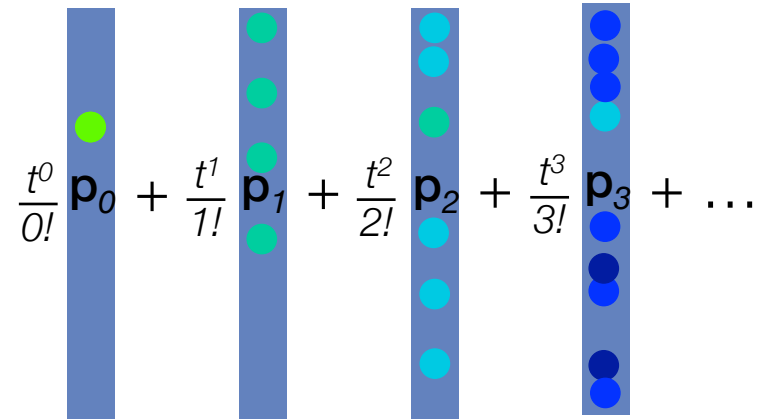
\mathbf{s} = normalized
seed vector

c_k = weight on
stage k

Heat Kernel vs. PageRank Diffusions

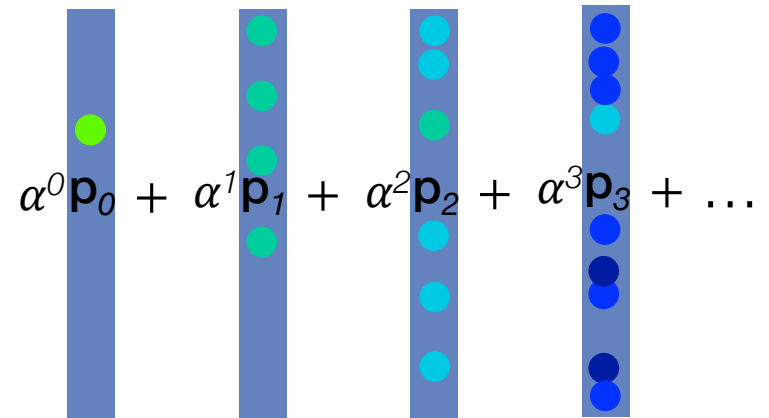
Heat Kernel uses $t^k/k!$

Our work is new analysis for this diffusion.



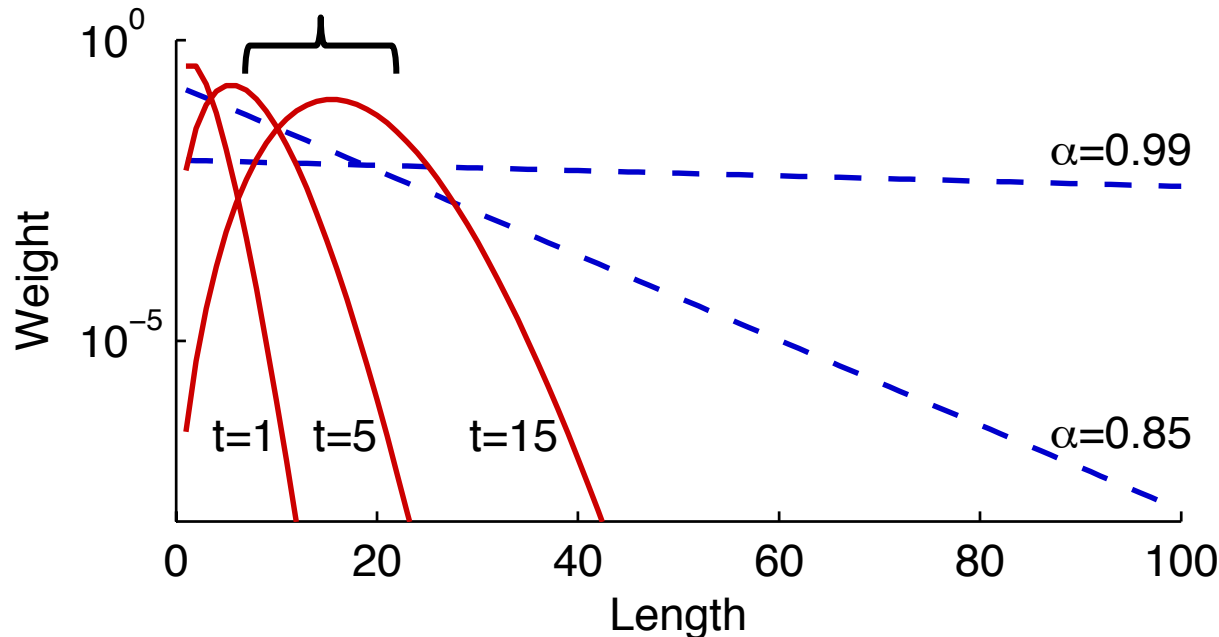
PageRank uses α^k at stage k.

Standard, widely-used diffusion we use for comparison.



Heat Kernel vs. PageRank Behavior

HK emphasizes **earlier stages** of diffusion.



PR, α^k

HK, $t^k/k!$

- involve **shorter walks** from seed,
- so **HK** looks at **smaller sets** than **PR**

Heat Kernel vs. PageRank Theory

good
conductance

fast
algorithm

PR

Local Cheeger Inequality:
“PR finds set of near-optimal conductance”

“PPR-push” is $O(1/(\epsilon(1-\alpha)))$
in theory, fast in practice
[Andersen Chung Lang 06]

HK

Heat Kernel vs. PageRank Theory

good
conductance

fast
algorithm

PR

Local Cheeger Inequality:
“PR finds set of near-
optimal conductance”

“PPR-push” is $O(1/(\epsilon(1-\alpha)))$
in theory, fast in practice
[Andersen Chung Lang 06]

HK

Local Cheeger Inequality
[Chung 07]

Heat Kernel vs. PageRank Theory

good
conductance

fast
algorithm

PR

Local Cheeger Inequality:
“PR finds set of near-optimal conductance”

“PPR-push” is $O(1/(\epsilon(1-\alpha)))$
in theory, fast in practice
[Andersen Chung Lang 06]

HK

Local Cheeger Inequality
[Chung 07]

Our work

Our work on **Heat Kernel**: theory

THEOREM Our algorithm for a relative ε -accuracy in a degree-weighted norm has

$$\text{runtime} \leq O(e^t(\log(1/\varepsilon) + \log(t)) / \varepsilon)$$

(which is constant, regardless of graph size)

Our work on **Heat Kernel**: theory

THEOREM Our algorithm for a relative ε -accuracy in a degree-weighted norm has
runtime $\leq O(e^t(\log(1/\varepsilon) + \log(t)) / \varepsilon)$
(which is constant, regardless of graph size)

COROLLARY **HK** is local!

($O(1)$ runtime \rightarrow diffusion vector has $O(1)$ entries)

Our work on **Heat Kernel**: results

First efficient, deterministic **HK** algorithm. Deterministic is important to be able to compare the behaviors of **HK** and **PR** *experimentally*:

Our key findings

- **HK** more accurately describes ground-truth communities in real-world networks
- identifies smaller sets → better precision
- speed & conductance comparable with **PR**



Twitter graph
41.6 M nodes
2.4 B edges

Python demo

un-optimized Python code
on a laptop

Available for download:

<https://gist.github.com/dgleich/cf170a226aa848240cf4>

Algorithm Outline

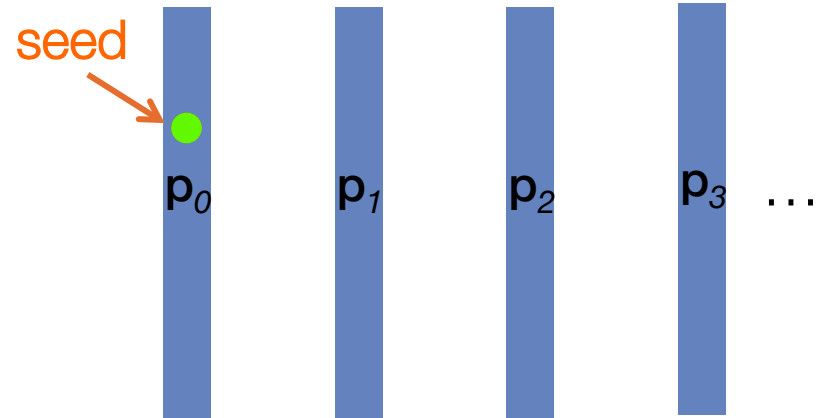
Computing **HK**

1. Pre-compute “**push**” thresholds
2. Do “**push**” on all entries above threshold

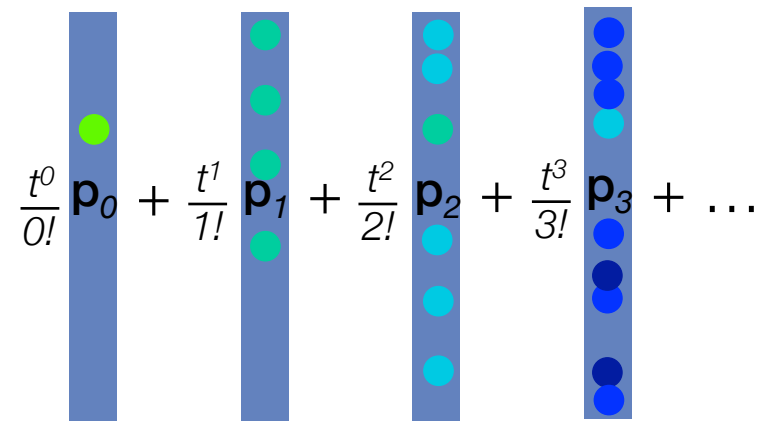
Algorithm Intuition

Computing **HK** given parameters t , ε , seed \mathbf{s}

Starting from here...



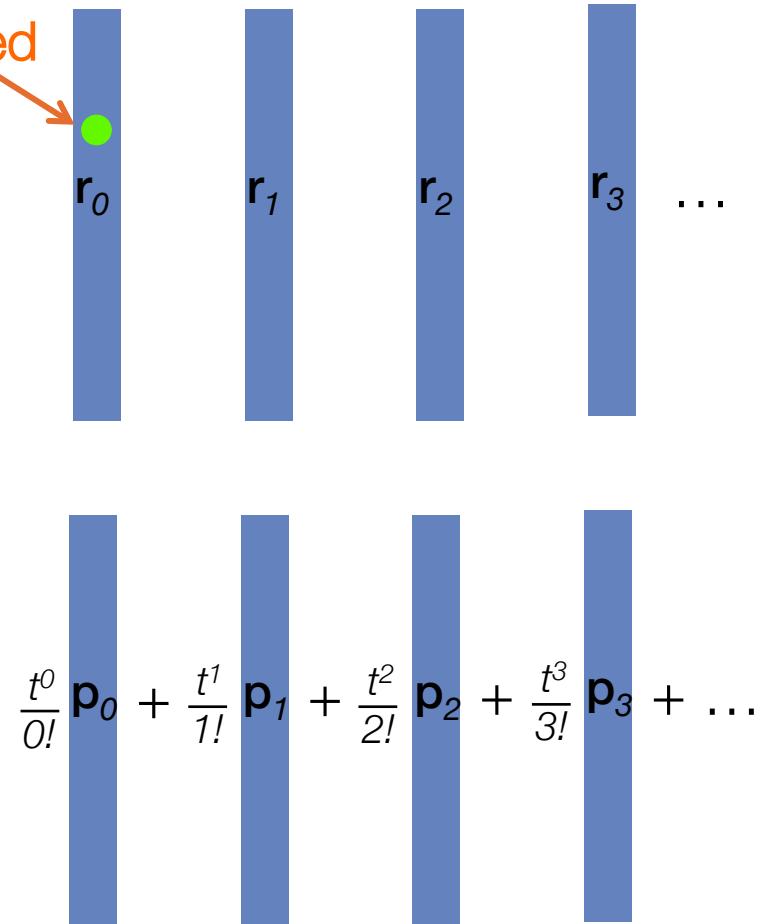
How to end up here?



Algorithm Intuition

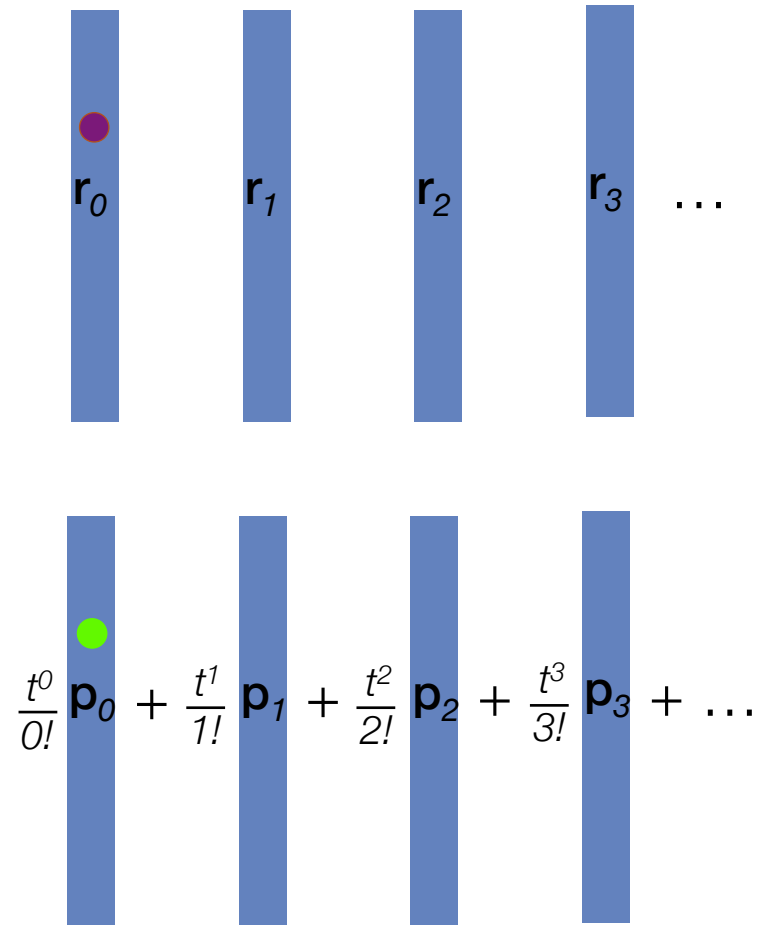
Begin with mass at seed(s)
in a “residual” staging area, r_0

The residuals r_k hold mass that
is unprocessed – it’s like *error*



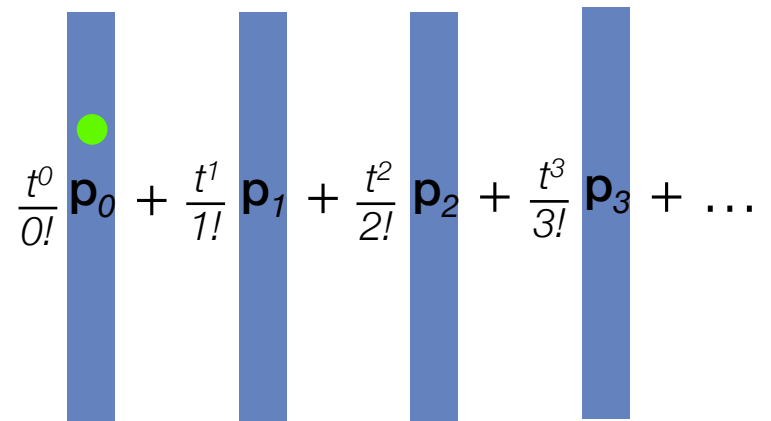
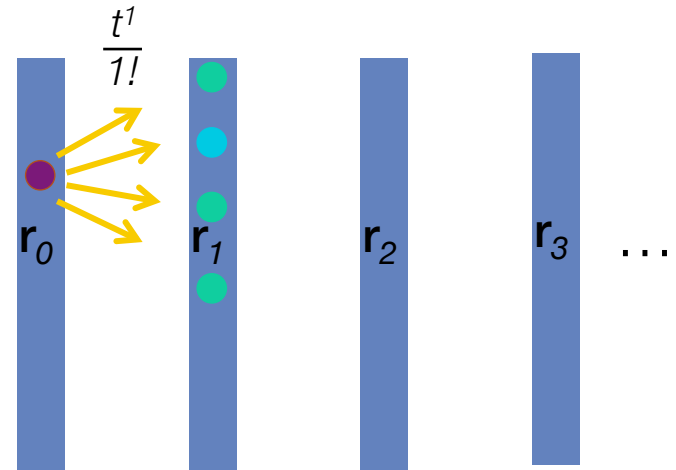
Push Operation

push – (1) remove entry in \mathbf{r}_k ,
(2) put in \mathbf{p} ,



Push Operation

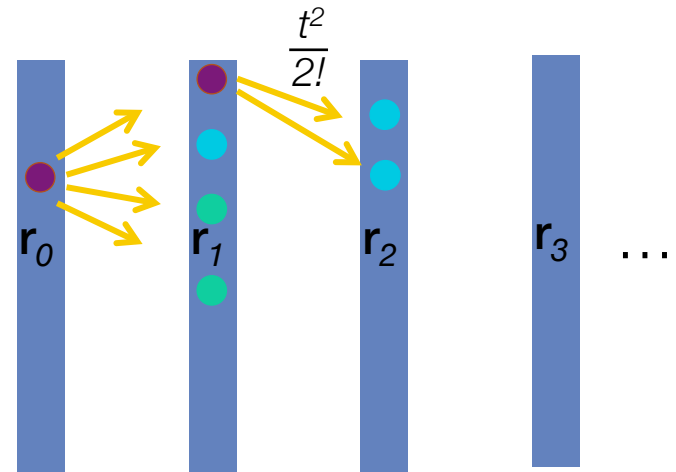
- push** – (1) remove entry in \mathbf{r}_k ,
(2) put in \mathbf{p} ,
(3) then scale and spread to neighbors in next \mathbf{r}



Push Operation

push – (1) remove entry in \mathbf{r}_k ,
(2) put in \mathbf{p} ,
(3) then scale and
spread to neighbors
in next \mathbf{r}

(repeat)



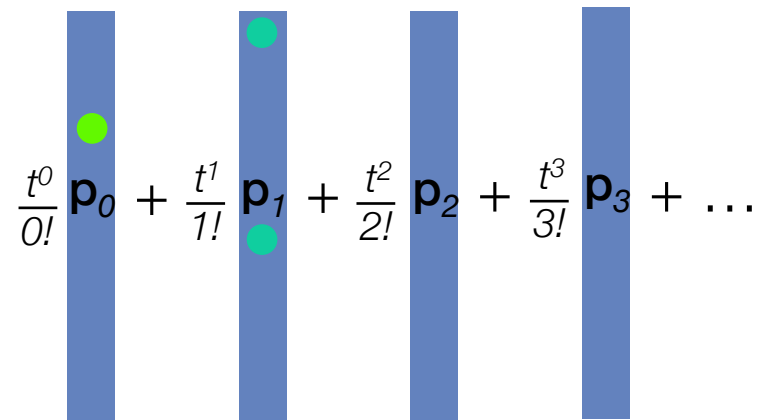
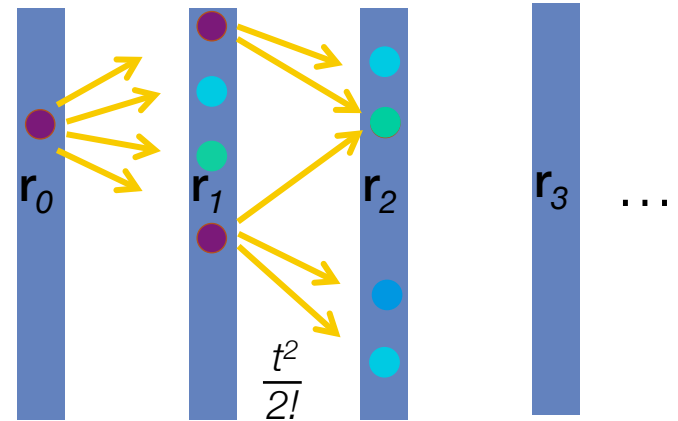
The diagram shows four vertical blue bars representing nodes $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ and an ellipsis. A green dot is in \mathbf{p}_0 . A green dot is in \mathbf{p}_1 . A green dot is in \mathbf{p}_2 . A green dot is in \mathbf{p}_3 . The labels $\frac{t^0}{0!}, \frac{t^1}{1!}, \frac{t^2}{2!}, \frac{t^3}{3!}$ are placed below the bars.

$$\frac{t^0}{0!} \mathbf{p}_0 + \frac{t^1}{1!} \mathbf{p}_1 + \frac{t^2}{2!} \mathbf{p}_2 + \frac{t^3}{3!} \mathbf{p}_3 + \dots$$

Push Operation

push – (1) remove entry in \mathbf{r}_k ,
(2) put in \mathbf{p} ,
(3) then scale and
spread to neighbors
in next \mathbf{r}

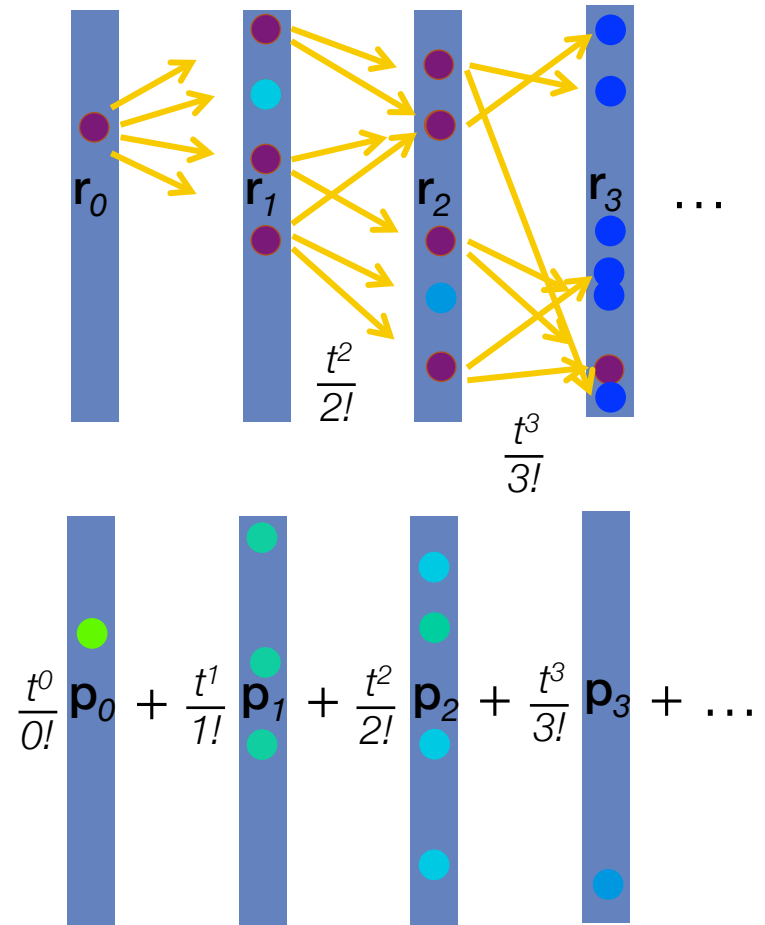
(repeat)



Push Operation

push – (1) remove entry in \mathbf{r}_k ,
(2) put in \mathbf{p} ,
(3) then scale and
spread to neighbors
in next \mathbf{r}

(repeat)

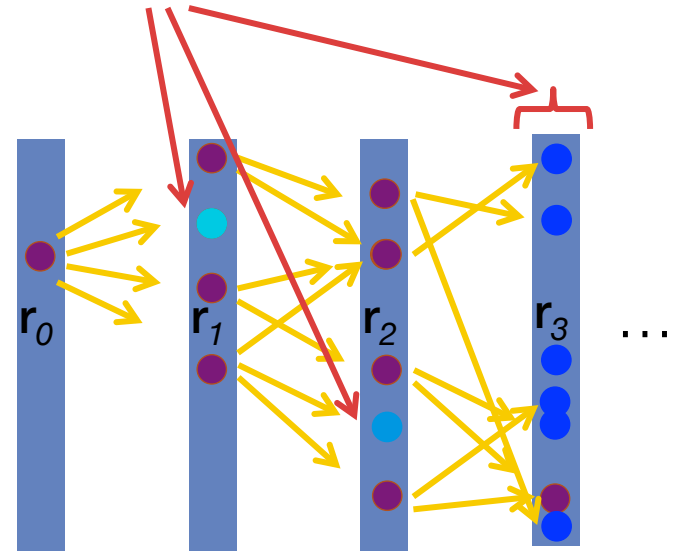


Thresholds

ERROR equals weighted sum of entries left in \mathbf{r}_k

→ Set threshold so “leftovers” sum to $< \varepsilon$

entries $<$ threshold



$$\frac{t^0}{0!} \mathbf{p}_0 + \frac{t^1}{1!} \mathbf{p}_1 + \frac{t^2}{2!} \mathbf{p}_2 + \frac{t^3}{3!} \mathbf{p}_3 + \dots$$

The diagram shows the weighted sum of vectors $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \dots$ represented as vertical blue bars. Each bar contains a single colored dot (green, cyan, blue). The weights are $\frac{t^0}{0!}, \frac{t^1}{1!}, \frac{t^2}{2!}, \frac{t^3}{3!}, \dots$.

Thresholds

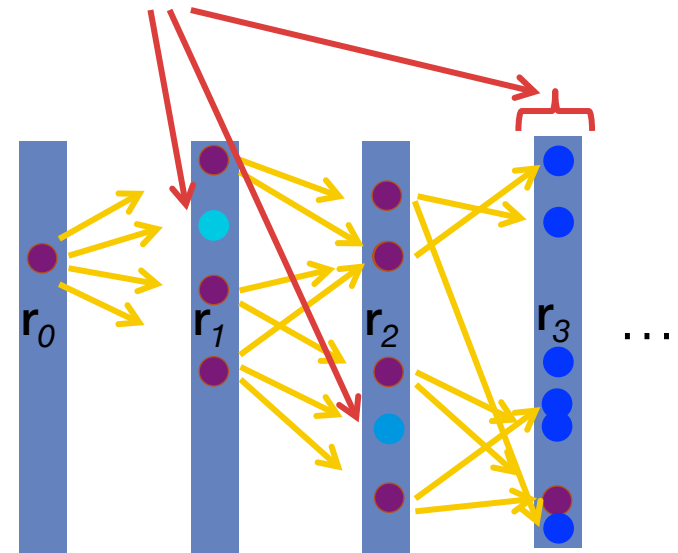
ERROR equals weighted sum of entries left in \mathbf{r}_k

→ Set threshold so “leftovers” sum to $< \varepsilon$

Threshold for stage \mathbf{r}_k is

$$\frac{\text{sum of remaining scale factors}}{\text{prior scaling factor}}$$

entries $<$ threshold



$$\frac{t^0}{0!} \mathbf{p}_0 + \frac{t^1}{1!} \mathbf{p}_1 + \frac{t^2}{2!} \mathbf{p}_2 + \frac{t^3}{3!} \mathbf{p}_3 + \dots$$

The diagram shows the expansion of the exponential function as a sum of vectors $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \dots$ with coefficients $\frac{t^k}{k!}$. Each vector is represented as a vertical blue bar with colored dots (green, cyan, blue) indicating the components of the vector.

Algorithm Outline

Computing **HK**

1. Pre-compute “**push**” thresholds
2. Do “**push**” on all entries above threshold

Communities in Real-world Networks

Given a seed in an unidentified real-world community, how well can **HK** and **PR** describe that community?

Measure quality using F_1 -measure.

Graph	$ V $	$ E $
amazon	330 K	930 K
dblp	320 K	1 M
youtube	1.1 M	3 M
lj	4 M	35 M
orkut	3.1 M	120 M
friendster	66 M	1.8 B

F_1 -measure

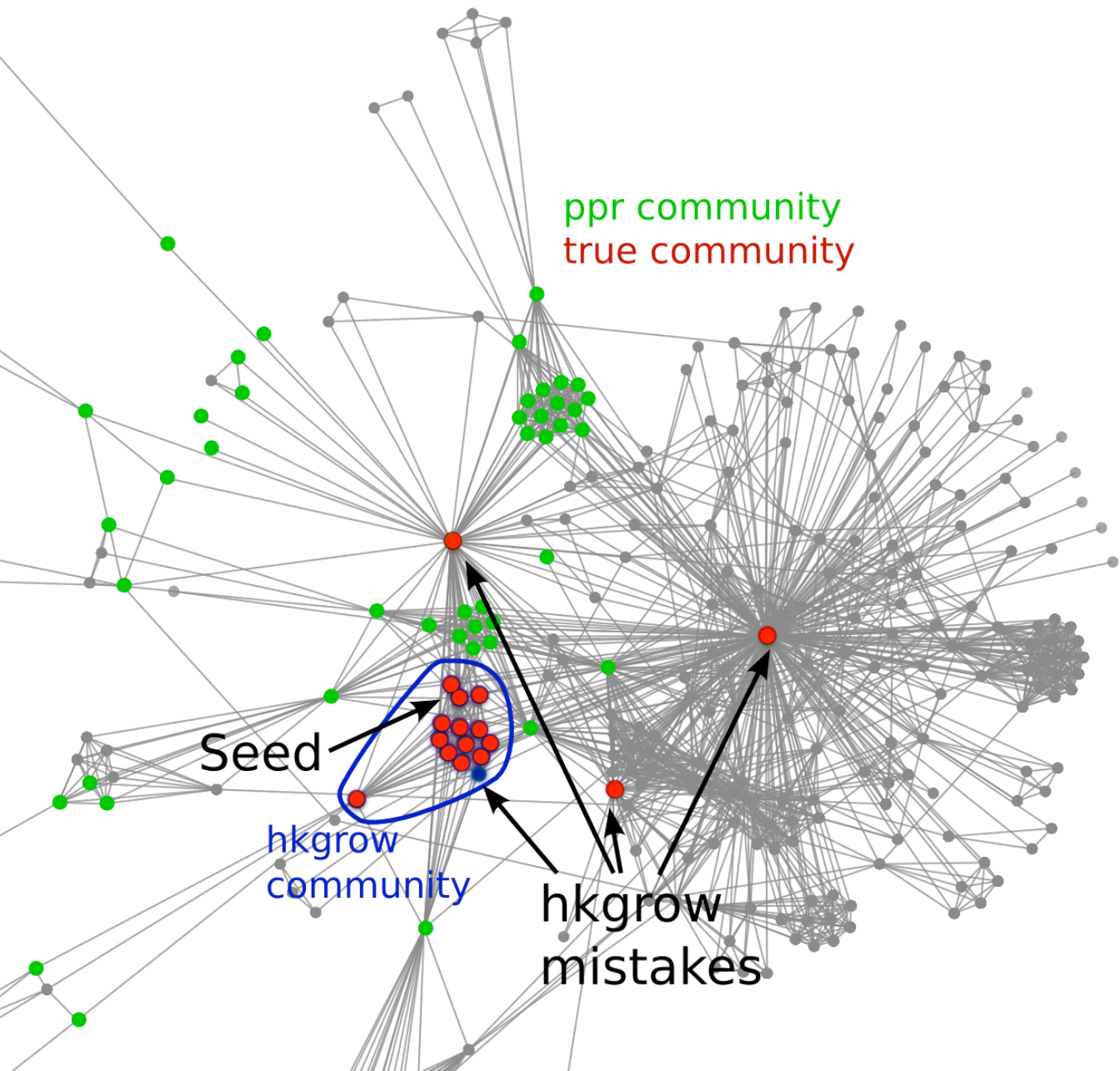
is the harmonic mean of

$$\text{precision} = \frac{\# \text{ correct guesses}}{\# \text{ total guesses}}$$

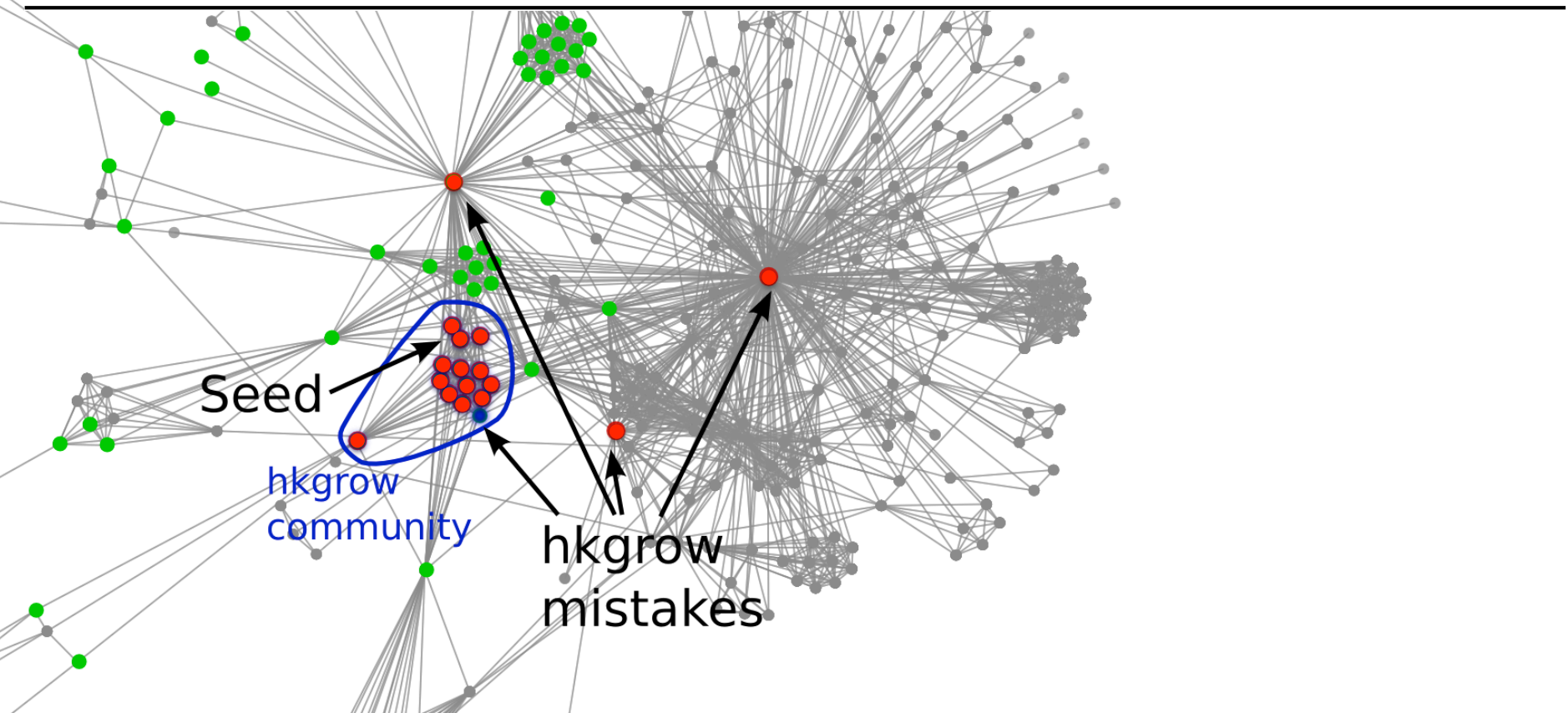
and

$$\text{recall} = \frac{\# \text{ answers you get}}{\# \text{ answers there are}}$$

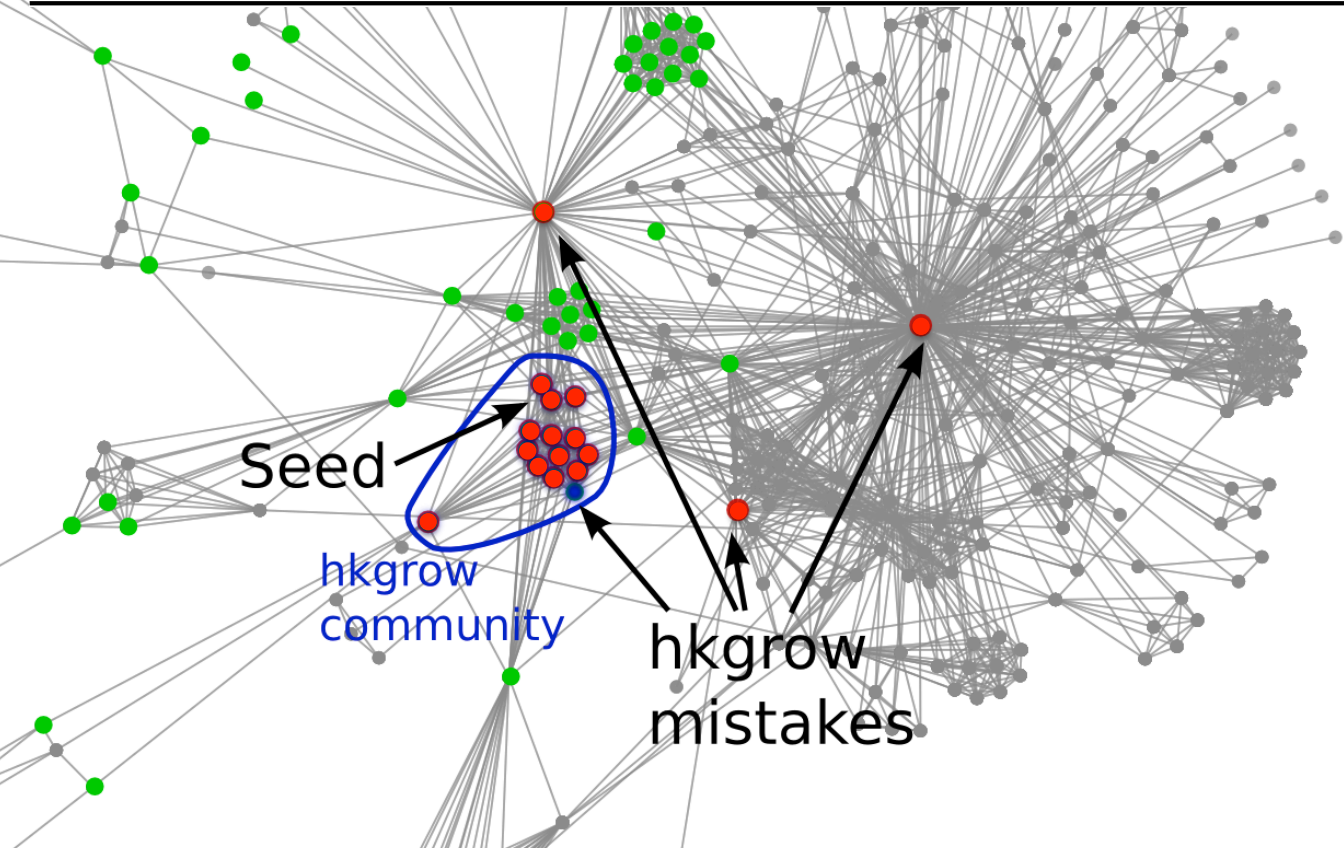
Datasets from SNAP collection [Leskovec]



data	F_1		precision		set size		comm size
	HK	PR	HK	PR	HK	PR	
amazon	0.325	0.140	0.244	0.107	193	15293	495



data	F_1		precision		set size		comm size
	HK	PR	HK	PR	HK	PR	
amazon	0.325	0.140	0.244	0.107	193	15293	495
dblp	0.257	0.115	0.208	0.081	44	16026	1429
youtube	0.177	0.136	0.135	0.098	1010	6079	1615
lj	0.131	0.107	0.102	0.086	283	738	662
orkut	0.055	0.044	0.036	0.031	537	1989	4526
friendster	0.078	0.090	0.066	0.075	229	333	724



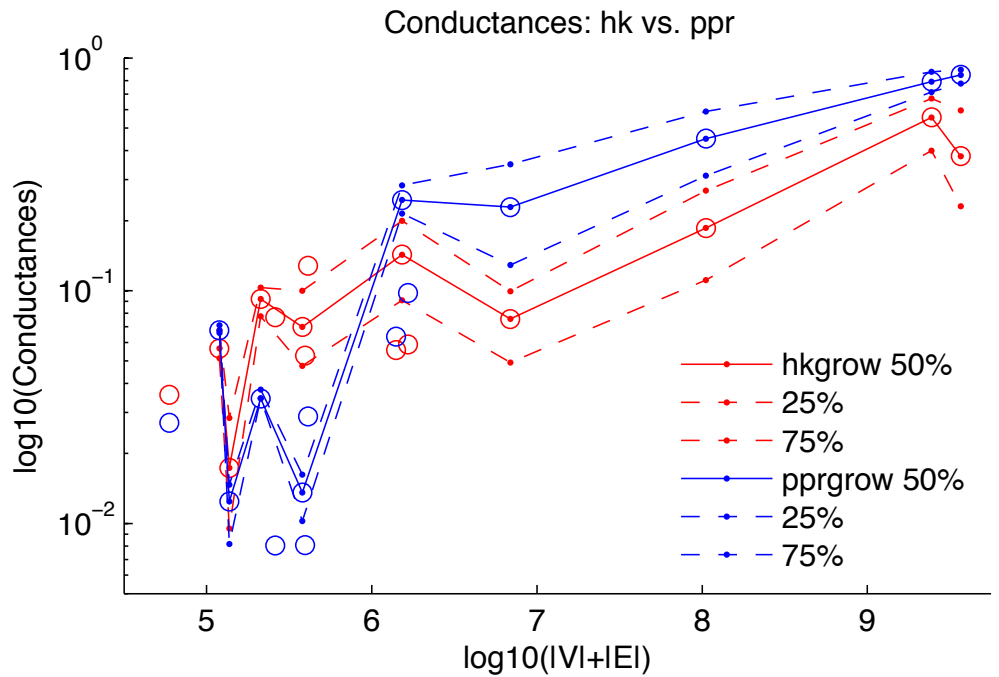
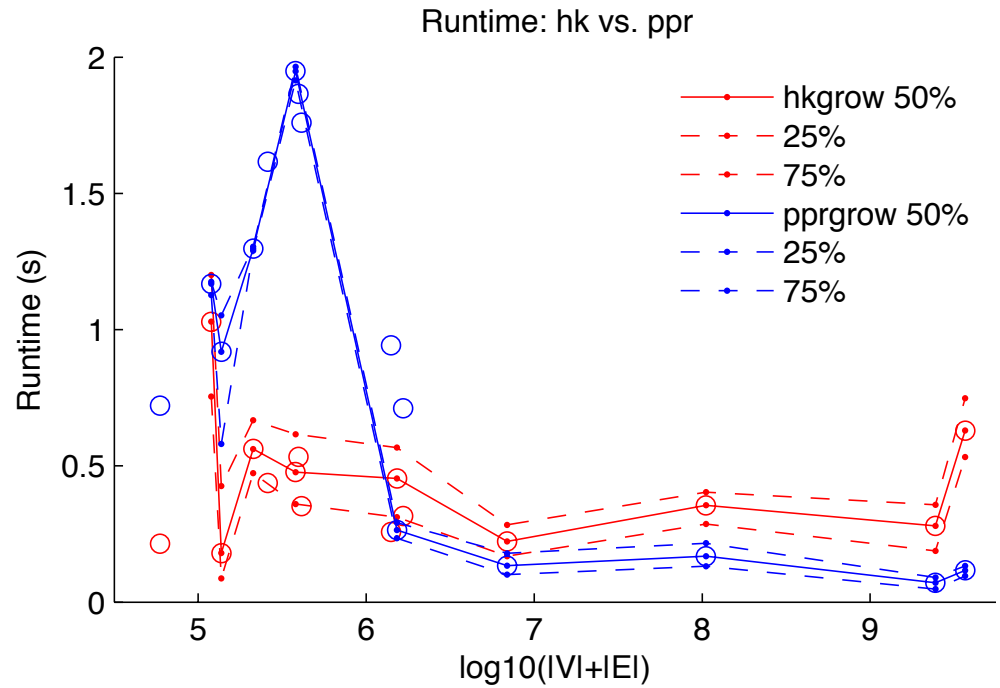
PR achieves high recall by “guessing” a huge set

HK identifies a tighter cluster, so attains better precision

Runtime & Conductance

HK is comparable in runtime and conductance.

As graphs scale, the diffusions' performance becomes even more similar.



Code, references, future work

Code available at

`http://www.cs.purdue.edu/homes/dgleich/codes/hkgrow`

Ongoing work

- generalizing to other diffusions
- simultaneously compute multiple diffusions

Questions or suggestions? Email Kyle Kloster at `kkloste-at-purdue-dot-edu`