Computing PageRank using MapReduce

Jasper Snoek

CSC2544 Final Project Report
Dec. 2008

Motivation

- The internet is huge: Google has found over 1 trillion unique urls[1]!

- Assume each url takes 0.5K, then we need over 400TB just to store URLs!

- Need an algorithm to rank webpages based on importance efficiently: **PageRank**

- Need a framework that allows the implementation PageRank in a distributed and highly scalable way: **MapReduce**

[1]http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

PageRank

The ranking of a page is determined by its estimated importance (determined by link structure) instead of by its content.

Sergey Brin and Lawrence Page (1998). "The anatomy of a large-scale hypertextual Web search engine". Proceedings of the seventh international conference on World Wide Web 7: 107-117

Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry (1999). The PageRank citation ranking: Bringing order to the Web

MapReduce

- “A programming model and associated implementation for processing and generating large data sets.” -J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. *OSDI,* 2004

- Many algorithms can be decomposed into two stages:

  1) A map stage that maps a key/value pair into intermediate sets of key/value pairs.

  2) A reduce stage that merges all of the values associated with the same key.

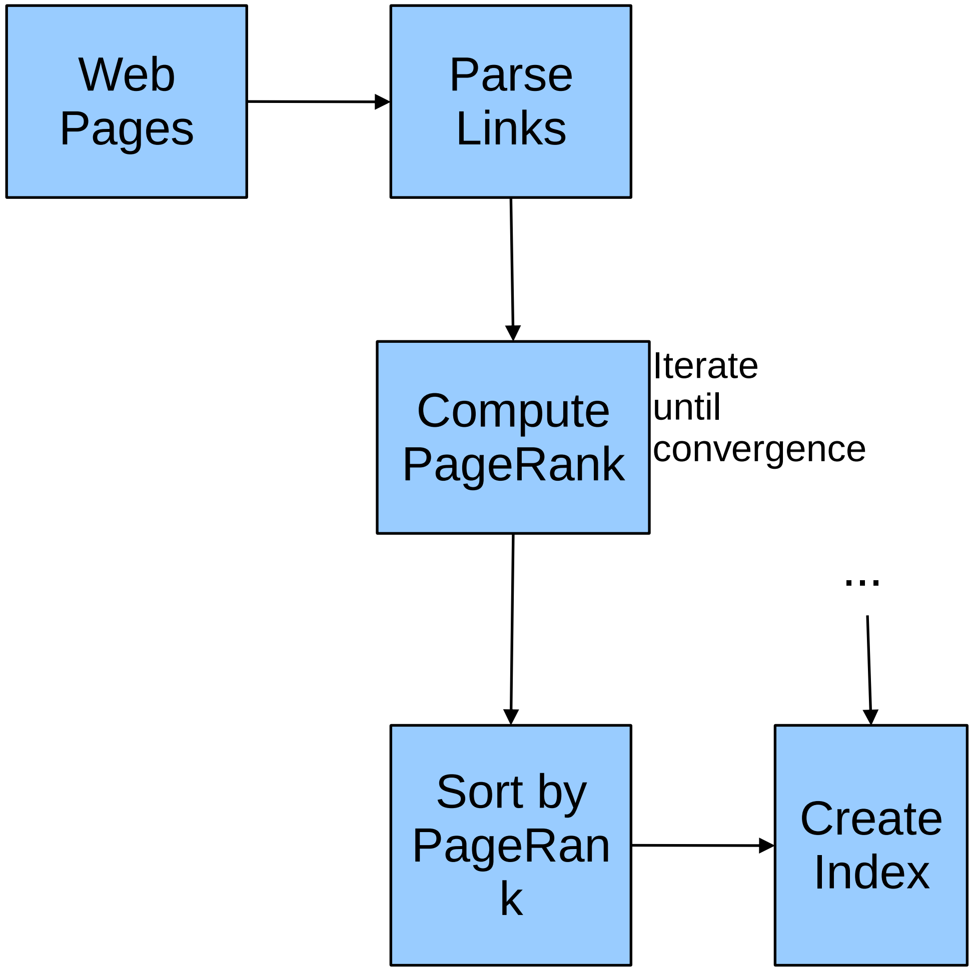
- Each stage is implemented as a separate function call for each key (running on a different thread, processor, or computer)

Computing PageRank using MapReduce

1) Parse documents (web pages) for links

2) Iteratively compute PageRank

3) Sort the documents by PageRank



Step 1: Parse URLs

- **Map:**
  - Input: index.html
    - **<html><body>Blah blah blah... A link....</html>**
  - Output for each out link:
    - key: “index.html”
    - value: “2.html”

- **Reduce:**
  - Input:
    - key: “index.html”
    - values: “2.html”, “3.html”, ...
  - Output:
    - key: “index.html”
    - Value: “1.0 2.html 3.html ...”

Start with a bunch of documents. Invoke a new Map call for each large chunk of a document.

Step 1: Parse URLs

- **Map:**
  - Input: index.html
    - **<html><body>Blah blah blah... A link....</html>**
  - Output for each out link:
    - key: "index.html"
    - value: "2.html"

For each link found, output the id (url) of the document as the key and the link as the value.

- **Reduce:**
  - Input:
    - key: "index.html"
    - values: "2.html", "3.html", ...
  - Output:
    - key: "index.html"
    - Value: "1.0 2.html 3.html ..."

Step 1: Parse URLs

- **Map:**
  - Input: index.html
    - **<html><body>Blah blah blah... A link....</html>**
  - Output for each out link:
    - key: "index.html"
    - value: "2.html"

- **Reduce:**
  - Input:
    - key: "index.html"
    - values: "2.html", "3.html", ...
  - Output:
    - key: "index.html"
    - Value: "1.0 2.html 3.html ..."

The reducer simply appends all the outlinks from a single document to a string and outputs <docid> <outlinks>

Step 1: Parse URLs

- **Map:**
  - Input: index.html
    - **<html><body>Blah blah blah... A link....</html>**
  - Output for each out link:
    - key: "index.html"
    - value: "2.html"

- **Reduce:**
  - Input:
    - key: "index.html"
    - values: "2.html", "3.html", ...
  - Output:
    - key: "index.html"
    - Value: "1.0 2.html 3.html ..."

We'll also assign an initial PageRank here. Start with a uniform PageRank for all pages.

Step 2: Page Rank Iterations

Map:

- Input:
  - key: index.html
  - value: <pagerank> 1.html 2.html...
  - Output for each outlink:
  - key: “1.html”
  - value: “index.html <pagerank> <number of outlinks>”

Start with the initial pagerank and outlinks of a document.

Reduce

- Input:
  - Key: “1.html”
  - Value: “index.html 0.5 23”
    Value: “2.html 2.4 2”
    Value: ...
- Output:
  - Key: “1.html”
  - Value: “<new pagerank> index.html 2.html...”

Step 2: Page Rank Iterations

Map:

- Input:
  - key: index.html
  - value: <pagerank> 1.html 2.html...
  - Output for each outlink:
  - key: “1.html”
  - value: “index.html <pagerank> <number of outlinks>”

For each outlink, output the docid of this document, its PageRank, and its total number of outlinks.

Reduce

- Input:
  - Key: “1.html”
  - Value: “index.html 0.5 23”
    Value: “2.html 2.4 2”
    Value: ...
- Output:
  - Key: “1.html”
  - Value: “<new pagerank> index.html 2.html...”

Step 2: Page Rank Iterations

Map:

- Input:
  - key: index.html
  - value: <pagerank> 1.html 2.html...
  - Output for each outlink:
  - key: “1.html”
  - value: “index.html <pagerank> <number of outlinks>”

Reduce

- Input:
  - Key: “1.html”
  - Value: “index.html 0.5 23”
    Value: “2.html 2.4 2”
    Value: ...
- Output:
  - Key: “1.html”
  - Value: “<new pagerank> index.html 2.html...”

Now the reducer has a document id, all the inlinks to that document and their corresponding PageRanks and number of outlinks.

Step 2: Page Rank Iterations

Map:

- Input:
  - key: index.html
  - value: <pagerank> 1.html 2.html...
  - Output for each outlink:
  - key: “1.html”
  - value: “index.html <pagerank> <number of outlinks>”

Reduce

- Input:
  - Key: “1.html”
  - Value: “index.html 0.5 23”
    Value: “2.html 2.4 2”
    Value: ...
- Output:
  - Key: “1.html”
  - Value: “<new pagerank> index.html 2.html...”

Compute the new PageRank and output in the same format as the URL parser.

Step 2: Page Rank Iterations

Map:

- Input:
  - key: index.html
  - value: <pagerank> 1.html 2.html...
  - Output for each outlink:
  - key: “1.html”
  - value: “index.html <pagerank> <number of outlinks>”

Reduce

- Input:
  - Key: “1.html”
  - Value: “index.html 0.5 23”
Value: “2.html 2.4 2”
Value: ...
- Output:
  - Key: “1.html”
  - Value: “<new pagerank> index.html 2.html...”

Now iterate until convergence!
Note: even computing convergence can be tricky when the computation is distributed.

Step 2: Computing PageRank

PageRank of document u

PageRank of document v that links to u

1 – damping factor

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

Damping factor

Number of outlinks from document v

Step 2: Computing PageRank

PageRank of document u

PageRank of document v that links to u

1 – damping factor

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

Damping factor

Number of outlinks from document v

Remember the reducer gets for each document *v* linking to *u:*

Key: "*u*"

Value: "*v* <PageRank of *v*> <number of outlinks from *v*>"

So we just sum over all the values passed to the reducer to compute the new PageRank!

Step 3: Sort Documents by PageRank

- **Last step:**
- Sort all the documents by pagerank
  - Assume many gigabytes of document ids, PageRanks and outlinks.
- MapReduce sorts all outputs by key using a distributed mergesort (very fast and scalable)
  - Sort the outputs from each reduce and then merge them to a file.
  - So output pagerank as key, document id as value and MapReduce takes care of the rest.

Map:

- Input:
  - Key: “index.html”
  - Value: “<pagerank> <outlinks>”
- Output:
  - Key: “<pagerank>”
  - Value: “index.html”

Implementation Details

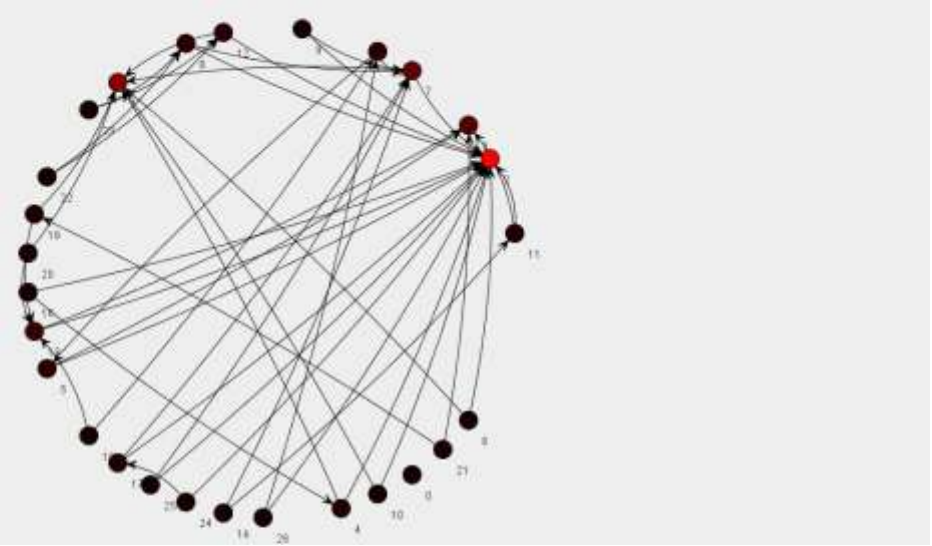
- Hadoop Open-Source MapReduce Framework
  - Java built on Apache
  - Used by Yahoo, Amazon
- No cluster – just my (dual-core) laptop :(

Testing the Algorithm

- Create a synthetic web graph

- JUNG (Java Universal Network/Graph) Framework
  - Open source Java library for modeling and viewing graphs

  - Generate random (semi-realistic) graphs and compute PageRank!
    - Note: I skipped the URL parsing stage here...



Testing the Algorithm: (Preferential Attachments)

Adding new vertices to the graph:

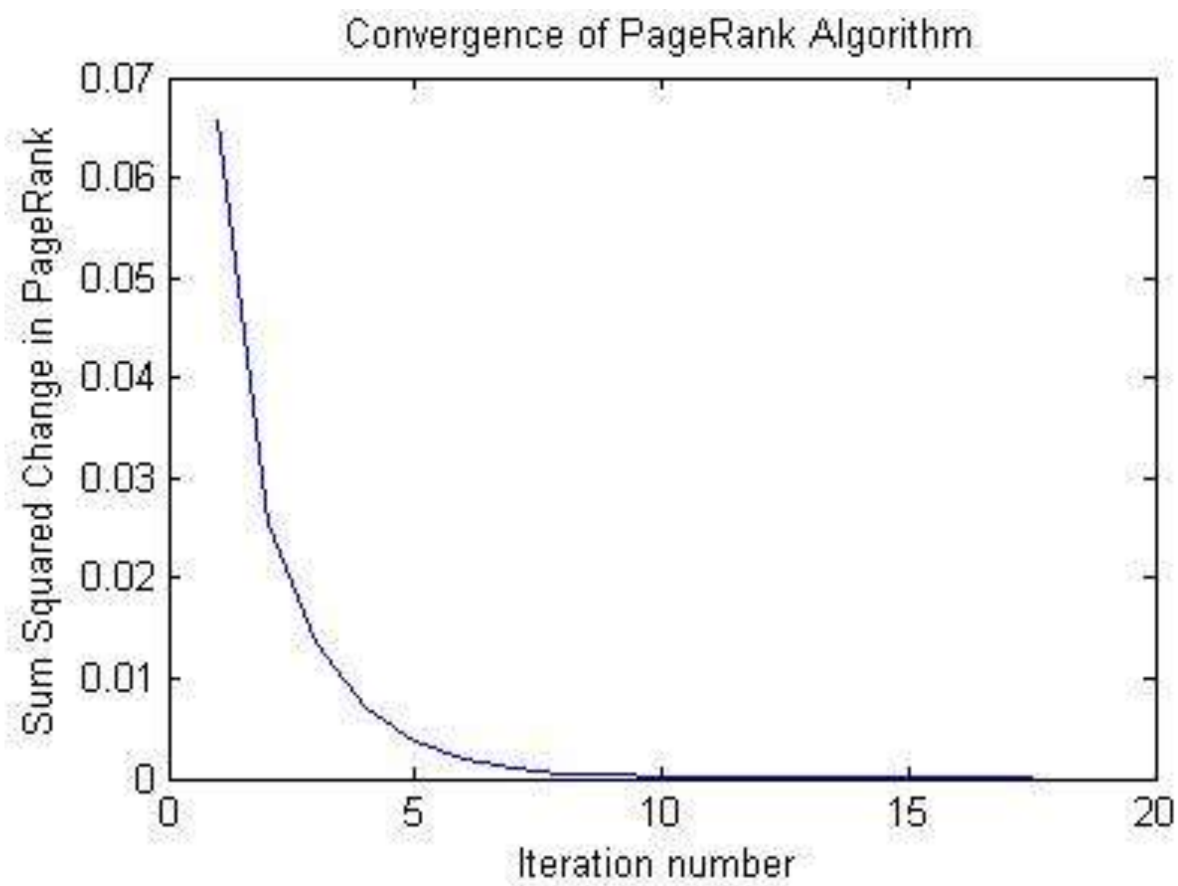
- A.-L. Barabasi and R. Albert, Emergence of scaling in random networks, Science 286, 1999.

- Create a new vertex and create *n* out-edges. Choose the targets of the out-edges with probability proportional to the in-degree of the target.

- Probability of adding an edge from a newly generated vertex to an existing vertex *v*:
  p = (in-degree(*v*) + 1) / (|E| + |V|)

  where |E| and |V| are the number of edges and vertices respectively.

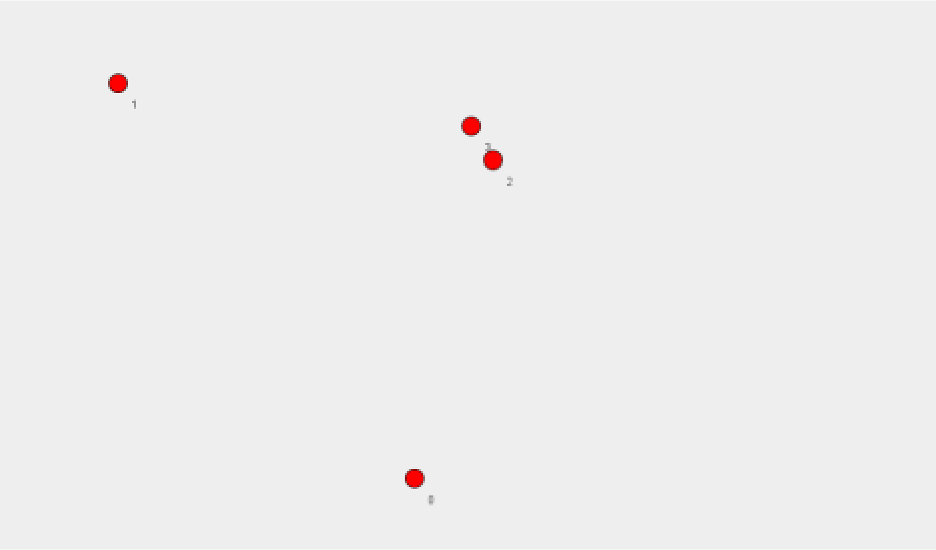
Testing the Algorithm



The PageRank algorithm converges rapidly for any sized web-graph. This figure shows the convergence for a graph of 1000 vertices (with an average of two links per vertex). Brin and Page report that PageRank computation for a webgraph of 322 million links converges in only 52 iterations. (Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry (1999). The PageRank citation ranking: Bringing order to the Web)

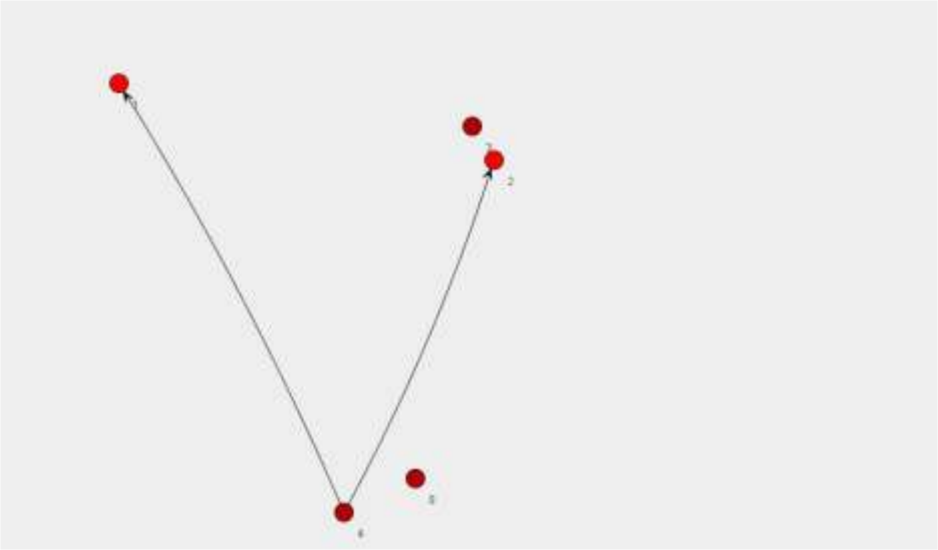
Generating a new graph

Start with a set of vertices



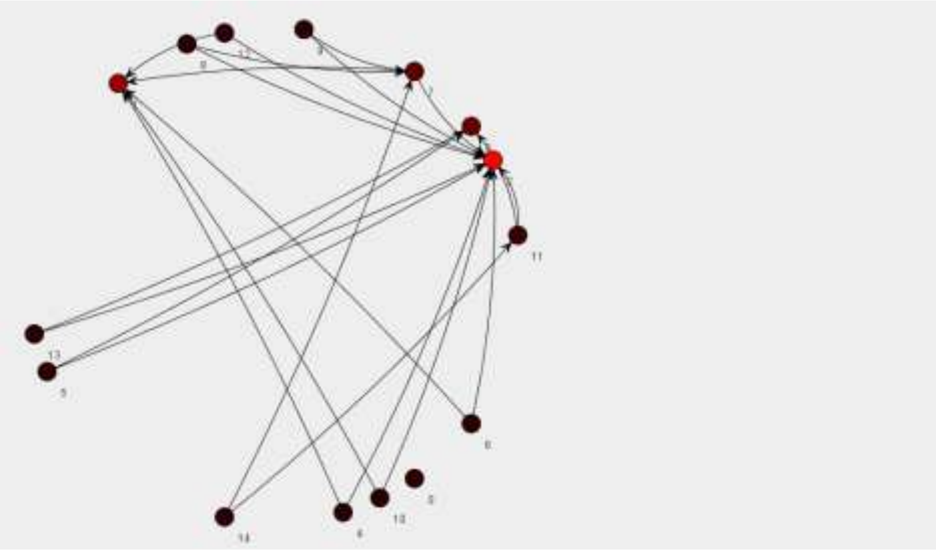
Add a new vertex with 2 links (targets are decided by the preferential attachment model)

(Vertices are colored by their relative PageRanks)

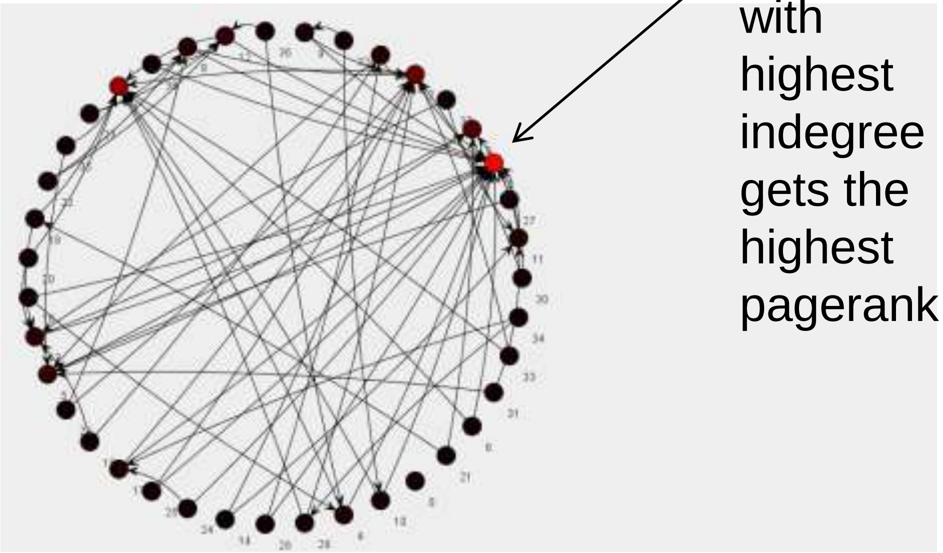


Generating a graph

After adding 10 new vertices



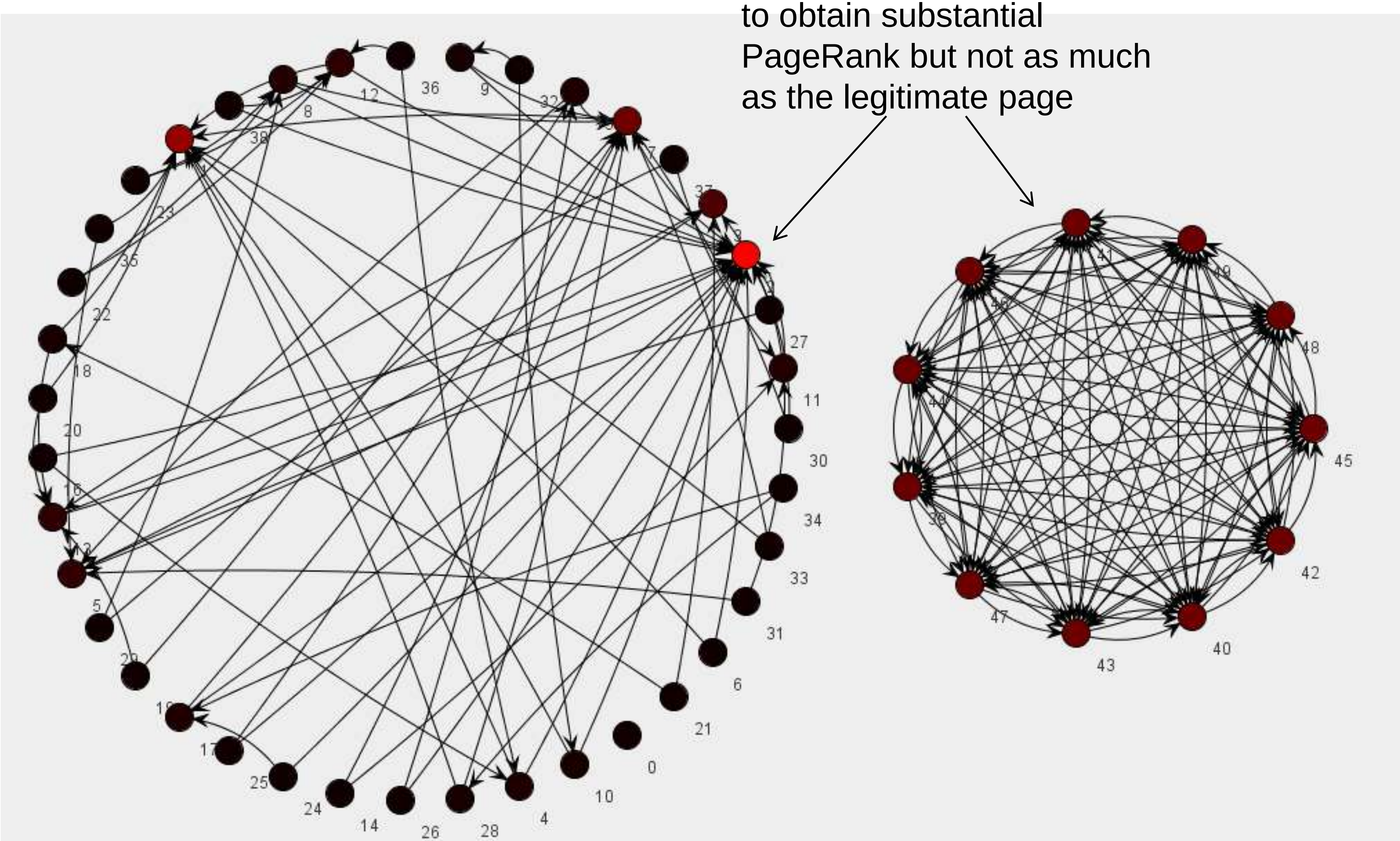
After adding 35 new vertices



The vertex with highest indegree gets the highest pagerank

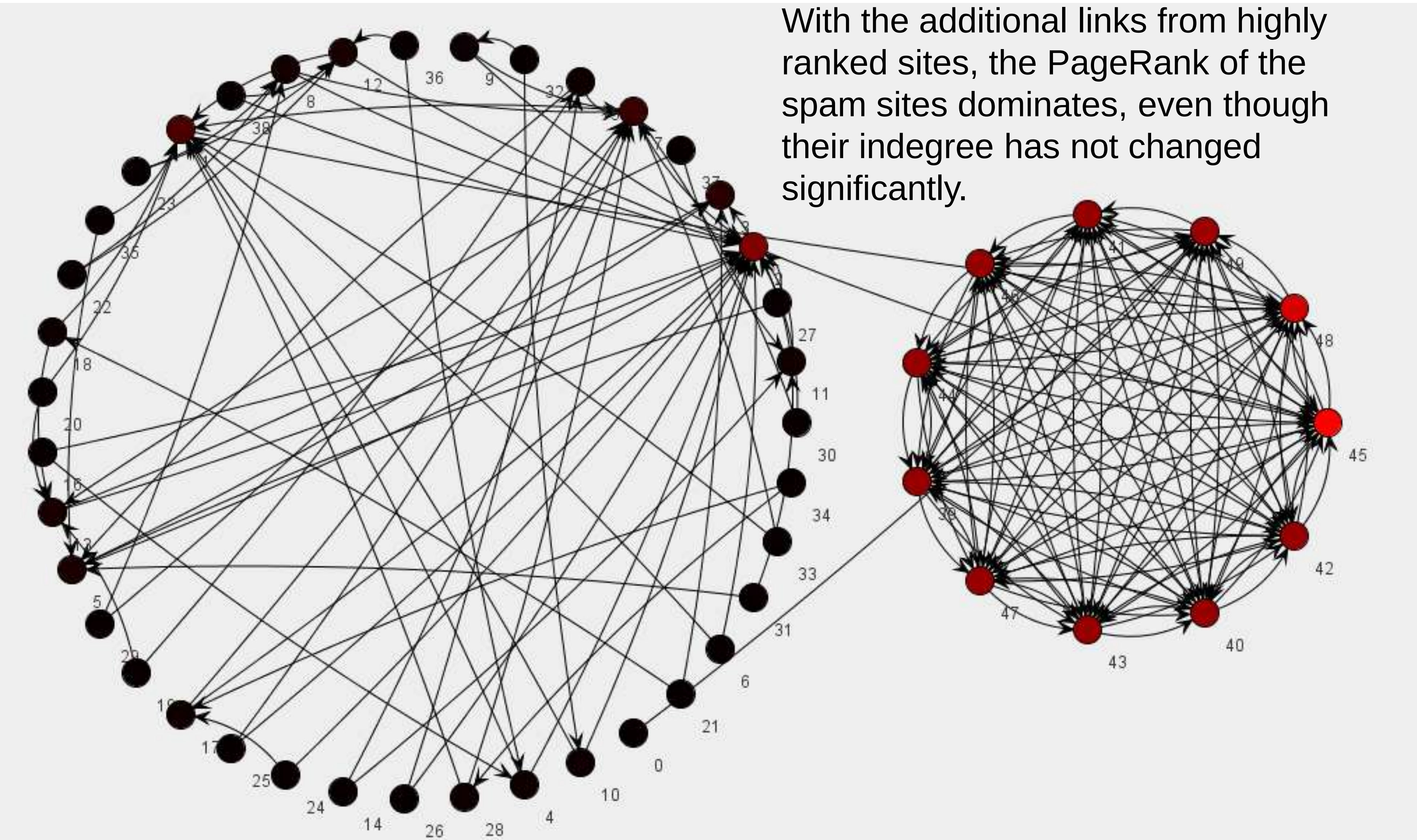
Someone created a densely connected spam network

The spam pages manage
to obtain substantial
PageRank but not as much
as the legitimate page



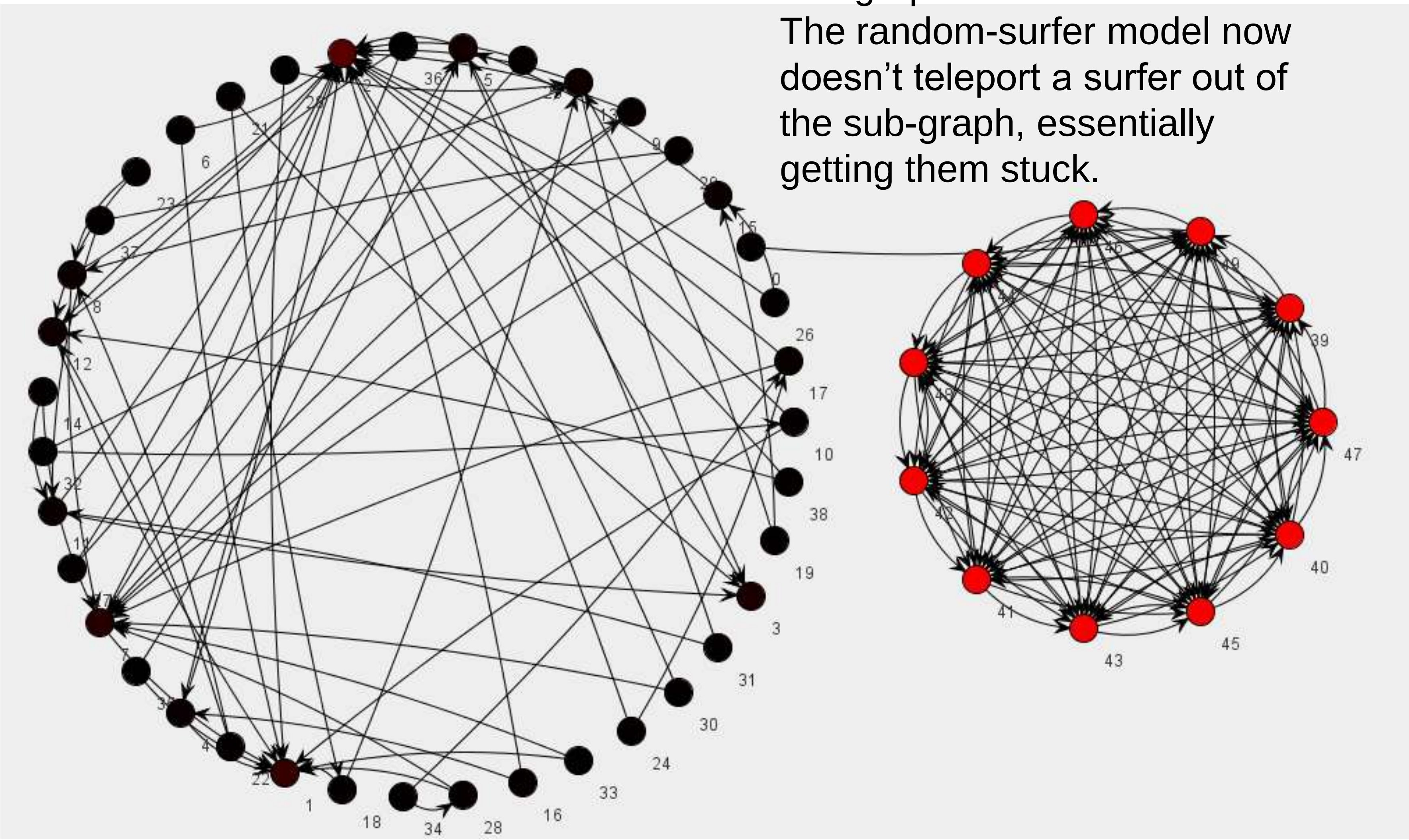
Now they hijacked a couple of web sites and added links to their spam sites!

With the additional links from highly ranked sites, the PageRank of the spam sites dominates, even though their indegree has not changed significantly.



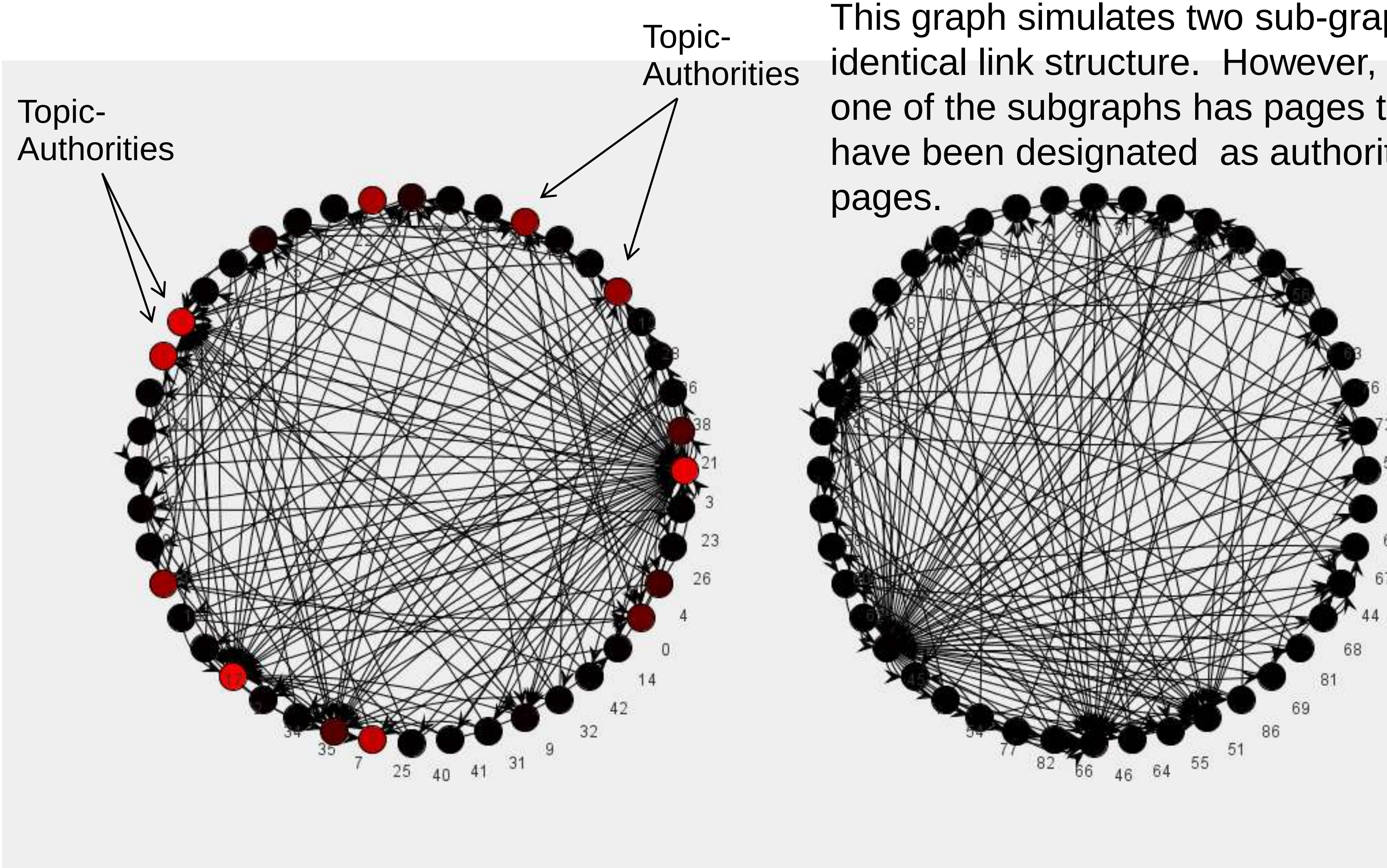
What if there's no decay term?

Now the densely connected sub-graph acts as a rank sink. The random-surfer model now doesn't teleport a surfer out of the sub-graph, essentially getting them stuck.



Topic-Based PageRank

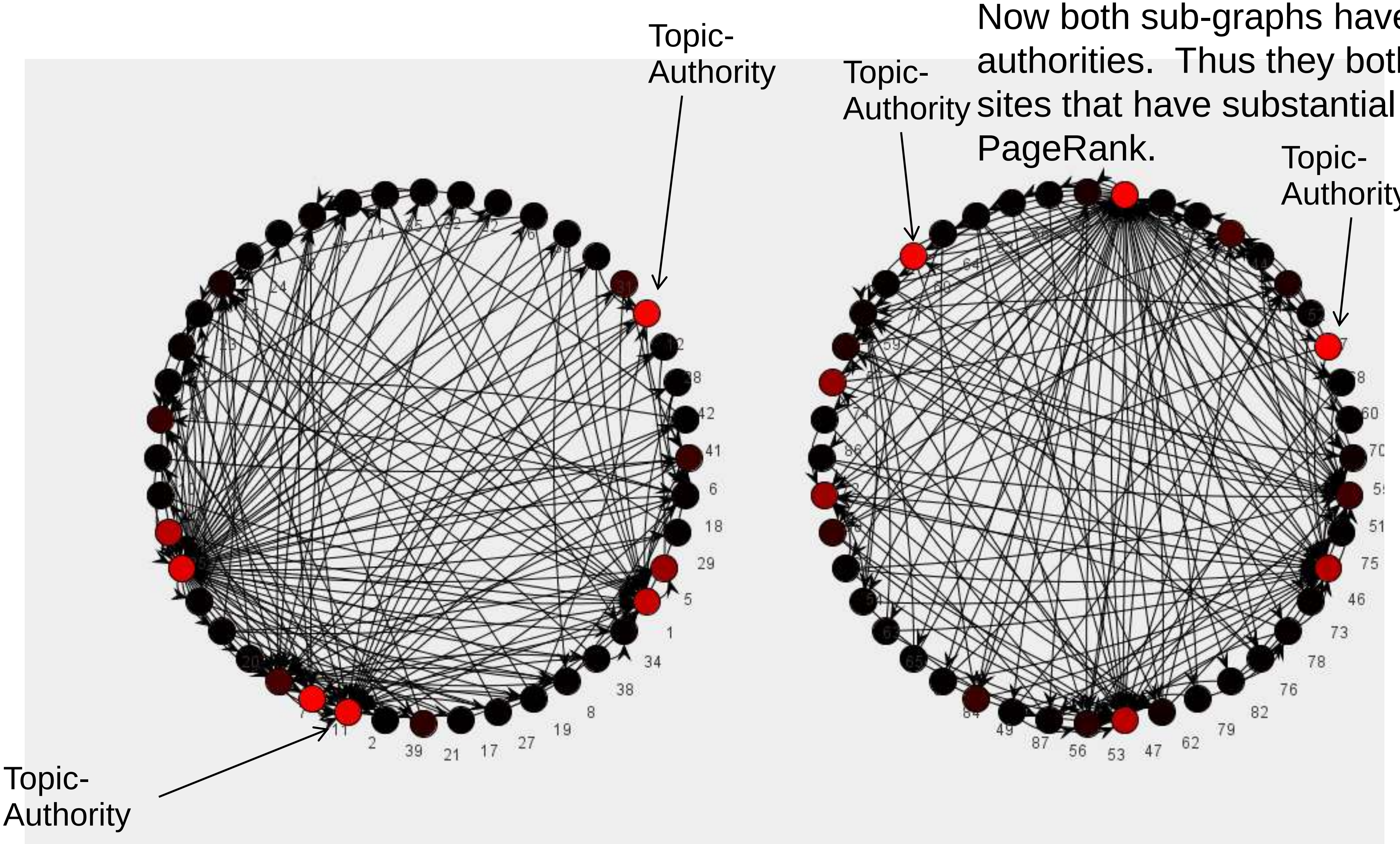
This graph simulates two sub-graphs with identical link structure. However, only one of the subgraphs has pages that have been designated as authority pages.



T. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. (2003) *IEEE Transactions on Knowledge and Data Engineering*

Topic-Based PageRank

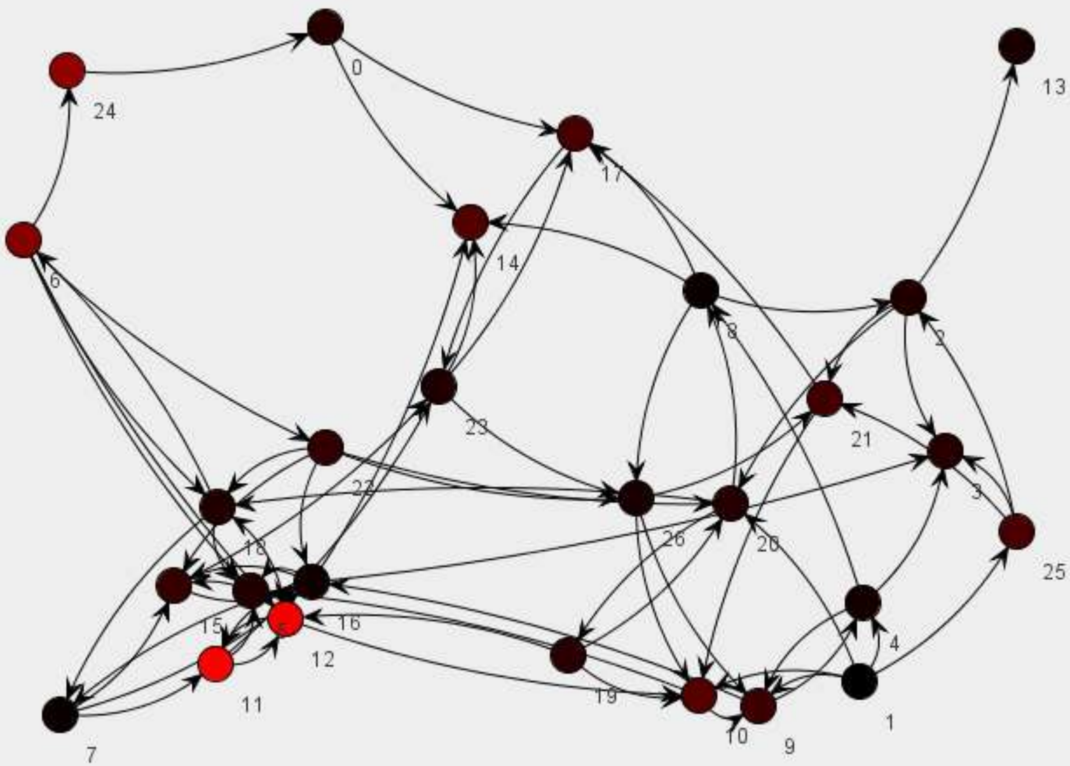
Now both sub-graphs have topic authorities. Thus they both have sites that have substantial PageRank.



T. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. (2003) *IEEE Transactions on Knowledge and Data Engineering*

A Randomly Connected Graph

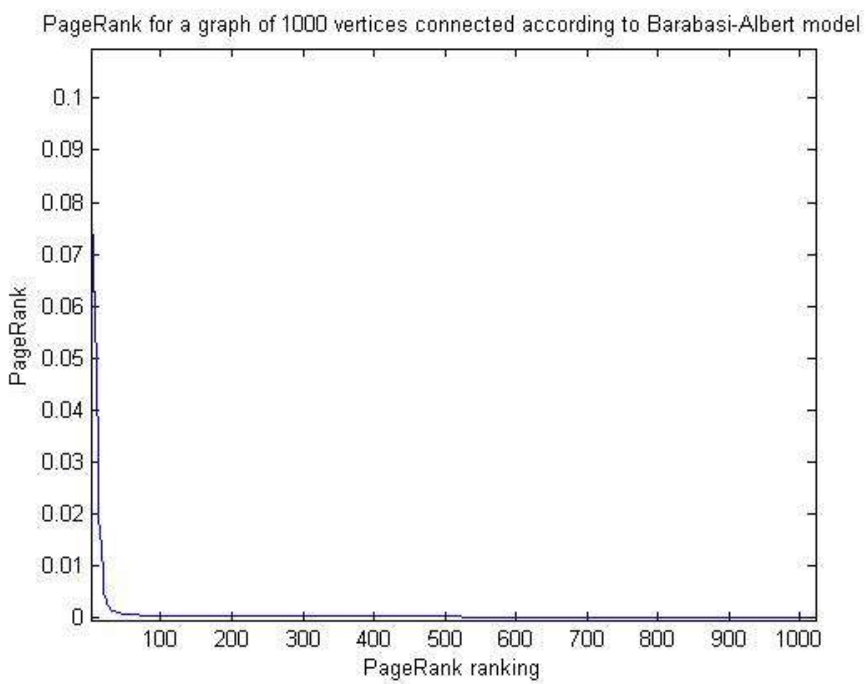
Interestingly, the distribution of PageRanks still appears to follow a power law, even when vertices are linked together completely randomly.

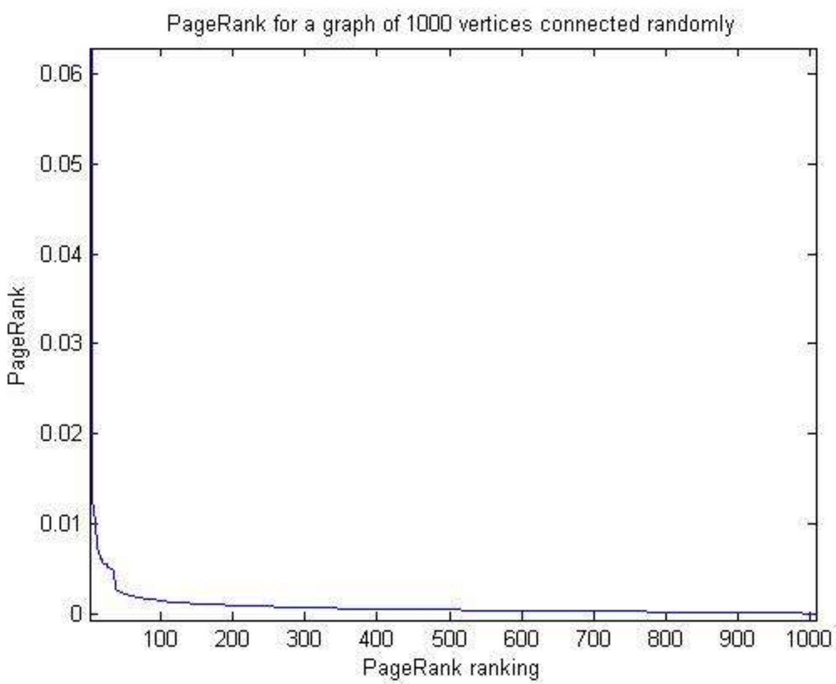


Distribution of PageRank

The distribution of PageRank for a graph where edges are connected according to the Barabasi-Albert model between 1000 vertices.

The distribution of PageRank for a graph where edges are connected randomly between 1000 vertices.





The distribution of PageRank closely resembles a power law with an exponent of approx. -1 as is reported in (N. Litvak, W. Scheinhardt and Y. Volkovich. Probabilistic Relation between In-Degree and PageRank). It is interesting that the distributions are similar for a randomly connected graph as well as a graph connected according to the Barabasi-Albert model. However, clearly the Barabasi-Albert model has a larger (i.e. more negative exponent).