NYU | TANDON SCHOOL OF ENGINEERING

Ranking, Mining, and Computing with Hyperlink Graphs

CS6913: Web Search Engines
CSE Department
NYU Tandon School of Engineering

Overview:

- Link-based ranking and related applications
  - Pagerank
  - HITS
  - Extensions and other applications
- Computing with large web graphs
  - Web graph compression and representation
  - Computational frameworks
  - Example: I/O-efficient Pagerank

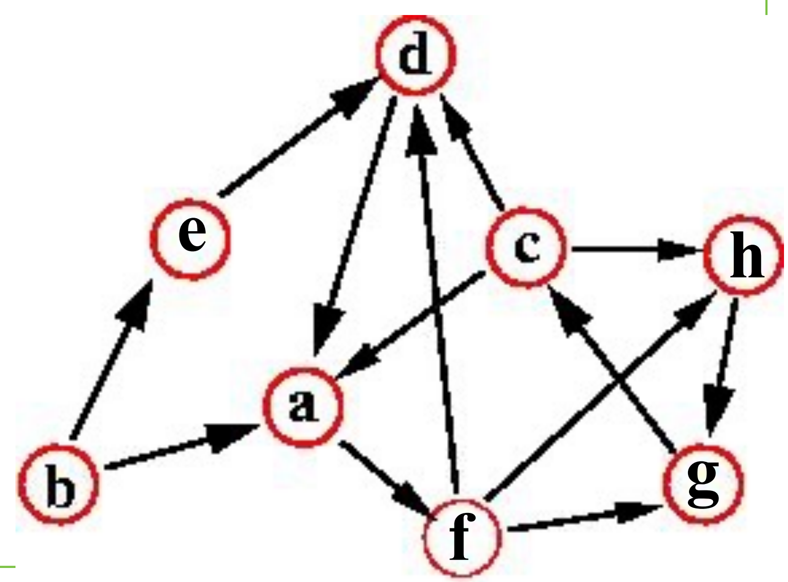
Link-Based Ranking Techniques

- Idea: use hyperlink structure as a feature for ranking
- Exploits/aggregates judgments by many content creators
- Many different methods, starting 1996-98
- Idea 1: “a link to a page is an endorsement of that page”
- Idea 2: “a link may indicate topical similarity”
- Idea 3: “links from or to the same other page indicate similarity”
- Idea 4: “linking to bad stuff reflects badly on you”
- We cover two algorithms: Pagerank and HITS
- Pagerank: global precomputation, query independent
- HITS: “hubs and authorities”, during query processing

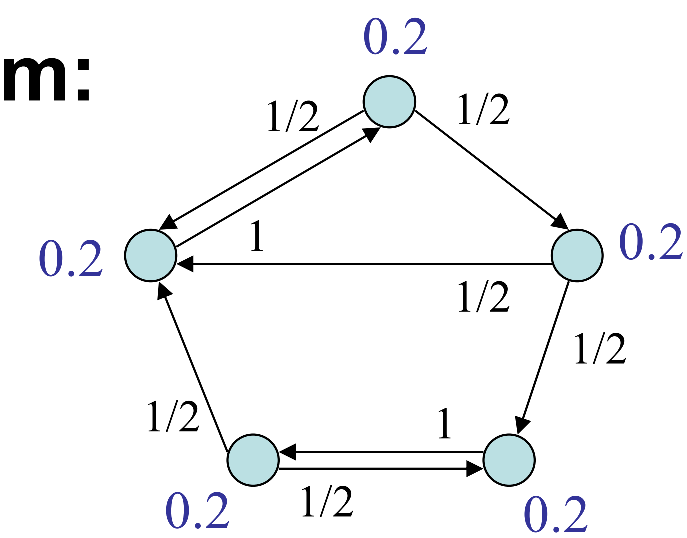
Pagerank Setup:

- Significance of a page is weighted sum of significance of pages linking to it
- Weighted by inverse of out-degree of pages pointing to page
- System of equation for the example graph:

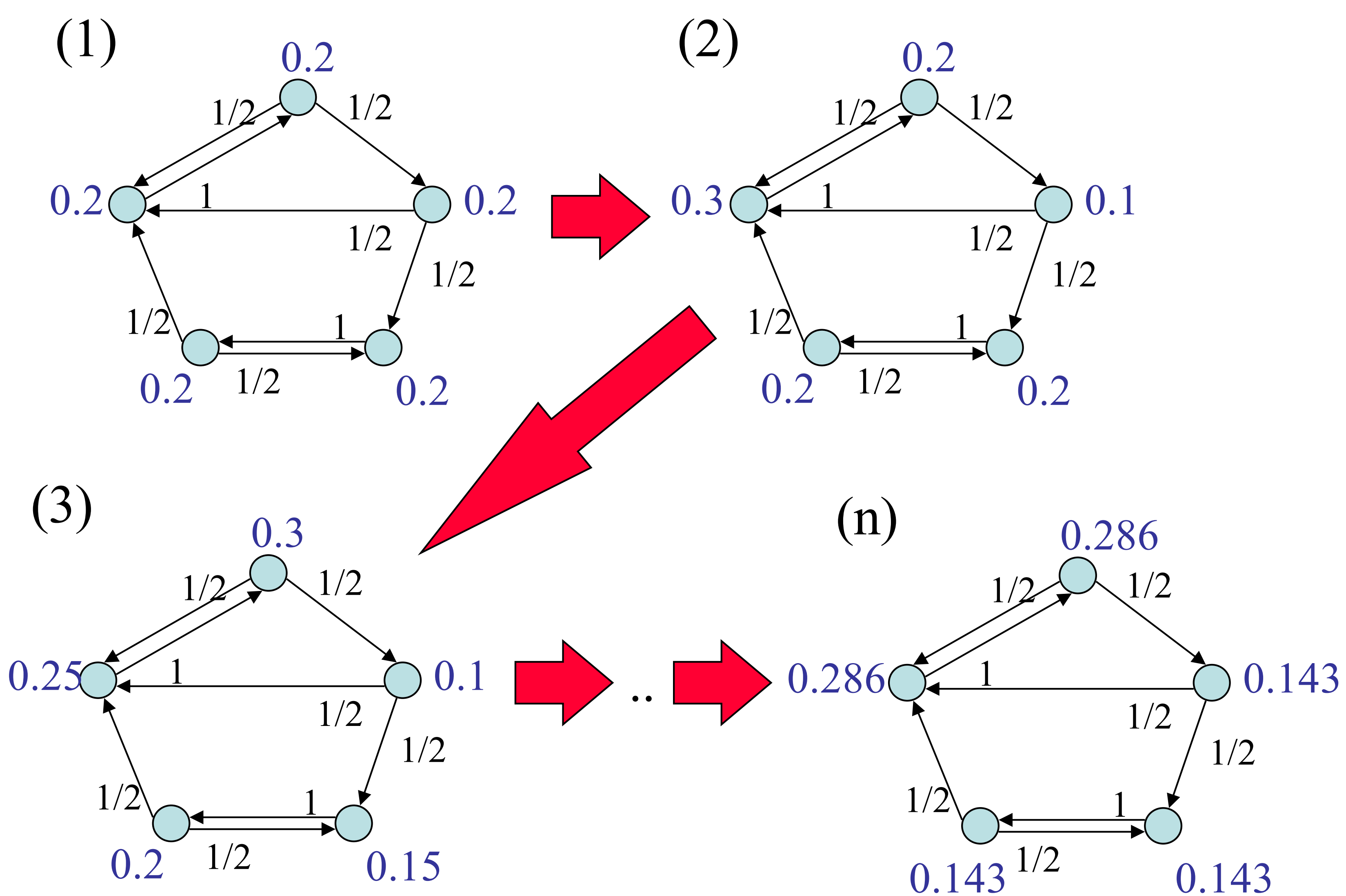
$$
\begin{aligned}
s(a) &= 1/2\, s(b) + 1/3\, s(c) + s(d)\\
s(b) &= 0\\
s(c) &= s(g)\\
s(d) &= 1/3\, s(c) + s(e) + 1/3\, s(f)\\
s(e) &= 1/2\, s(b)\\
s(f) &= s(a)\\
s(g) &= 1/3\, s(f) + s(h)\\
s(h) &= 1/3\, s(c) + 1/3\, s(f)
\end{aligned}
$$



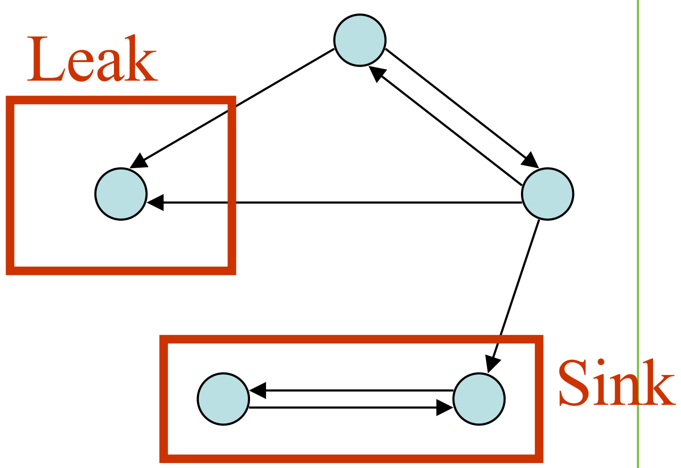
Iterative Pagerank Algorithm:



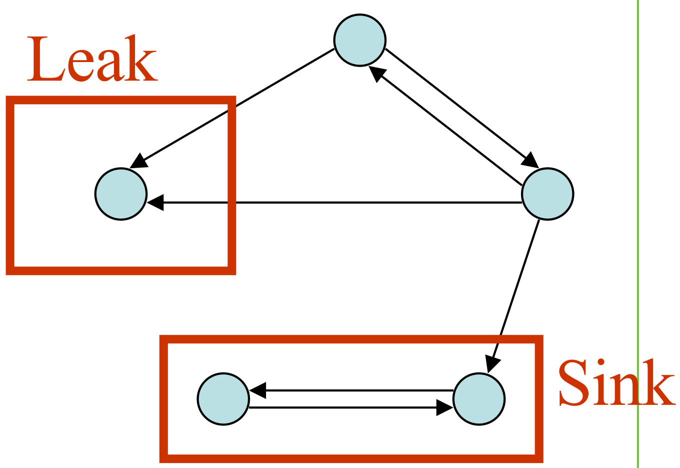
- initialize the *rank value* of each node to *1/n* (0.2 for 5 nodes)
- a node with *k* outgoing links transmits a *1/k* fraction of its current *rank value* over that edge to its neighbor
- iterate this process many times until it converges
- NOTE: this is a random walk on the link graph
- Pagerank: stationary distribution of this random walk


(1)
0.2
1/2
1/2
0.2
1
0.2
1/2
1/2
1/2
1
0.2
1/2
0.2
(2)
0.2
1/2
1/2
0.3
1
0.1
1/2
1/2
1/2
1
0.2
1/2
0.2
(3)
0.3
1/2
1/2
0.25
1
0.1
1/2
1/2
1/2
1
0.2
1/2
0.15
..
(n)
0.286
1/2
1/2
0.286
1
0.143
1/2
1/2
1/2
1
0.143
1/2
0.143

Convergence

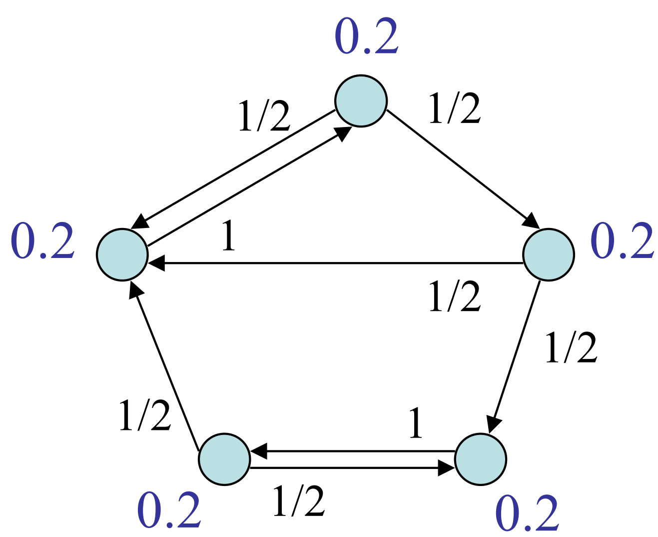


Convergence



- dealing with leaks:
  - prune away leaks, or
  - add back links, or
  - always do random jump from leak
- dealing with sinks
  - add a random jump from all nodes to escape sink
  - e.g., with probability *b = 0.15* jump to random node
- goal: after we deal with leaks and sinks, Pagerank will hopefully converge to a unique solution!

Matrix Notation



$$A = \begin{bmatrix} 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- stationary distribution: vector x with $xA = x$
- A is primitive, and x eigenvector of A
- computed using Jacobi or Gauss Seidel iteration
- note: above matrix is without the random jump

Matrix Notation with Random Jump

- suppose you add random jump with b = 0.15
- each "unit" of pagerank jumps to random node with probability 0.15, and follows random link with 0.85

Matrix Notation with Random Jump

- suppose you add random jump with b = 0.15
- each "unit" of pagerank jumps to random node with probability 0.15, and follows random link with 0.85
- this changes the matrix as follows:
  - multiple each value in matrix by 0.85
  - and then add 0.15/n to every entry (n = number of nodes)
  - A' = 0.85 * A + [0.15/n] (where [x] is a matrix of x values)
- then we find eigenvector for this matrix
- can use same iterative algorithm to do so

Three Views of Pagerank

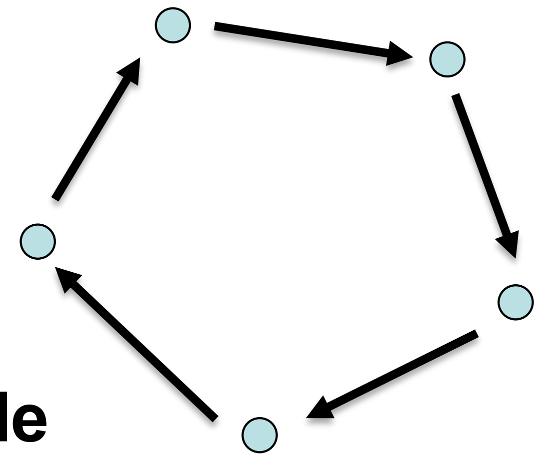
- 1. iterative algorithm
- 2. matrix notation and eigenvector problem
- 3. random walk on the graph (random surfer model)
- these are of course equivalent
- be skeptical about intuitive stories (e.g., random surfer)
- question in the end: does it work?
- to a degree, at some point, with suitable changes

Convergence Properties of Pagerank

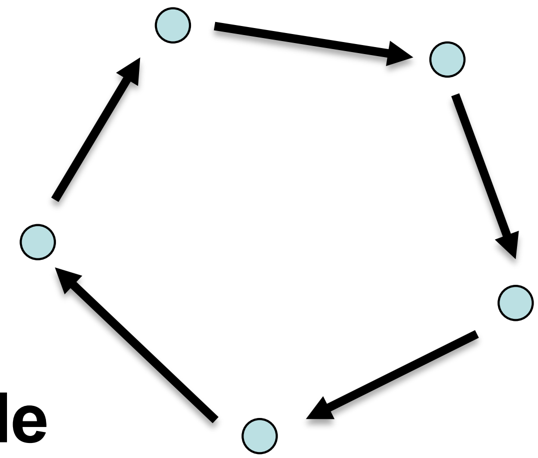
- So, does Pagerank always converge?
- Is there a unique solution, independent of initial?
- Only if we deal with leaks, sinks, and periodicity!
- Random jump takes care of those issues
- Let's go a little deeper into this!

Periodicity in Graphs

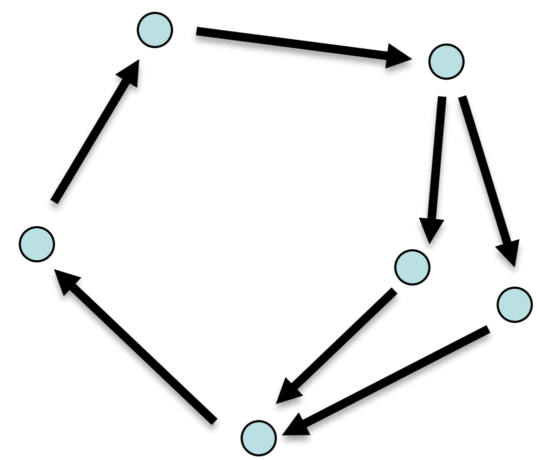
- consider a circle graph
- irreducible, not primitive
- suppose initially all PR in one node
- then PR will circulate forever



Periodicity in Graphs



- consider a circle graph
- irreducible, not primitive
- suppose initially all PR in one node
- then PR will circulate forever



- suppose equal initial assignment
- graph on right will not converge

Primitive and Periodic Matrices

- a matrix A is irreducible if for every i, j there exists a value m such that entry i, j of A^m is > 0
- this means the graph is strongly connected, since for any two nodes there exists a path from i to j
- a matrix A is primitive if there exists a value m such that every entry in A^m is > 0
- this means the graph is strongly connected and also aperiodic
- random jump makes the matrix primitive for m = 1

Primitive and Periodic Matrices

- random jump makes the matrix primitive for m = 1
- recall: random jump changes the matrix as follows:
  - multiple each value in matrix by 0.85
  - and then add 0.15/n to every entry (n = number of nodes)
- this makes every entry > 0
- so we can go between any two nodes in one step (with a small probability)

Theory: Convergence of Pagerank

- given a matrix A, the spectral radius R(a) is defined as max | ev_i(A) | where ev_i(A) are the eigenvals of A

Theory: Convergence of Pagerank

- given a matrix A, the spectral radius R(a) is defined as max | ev_i(A) | where ev_i(A) are the eigenvals of A
- Perron's Theorem: For a primitive matrix A:
  - there exists an eigenvalue R(a) with multiplicity 1
  - any other eigenvalue v has | v | < R(a)

Theory: Convergence of Pagerank

- given a matrix A, the spectral radius R(a) is defined as max | ev_i(A) | where ev_i(A) are the eigenvals of A
- Perron's Theorem: For a primitive matrix A:
  - there exists an eigenvalue R(a) with multiplicity 1
  - any other eigenvalue v has | v | < R(a)
- also, $\min_i r_i$ <= R(a) <= $\max_i r_i$ (where r_i is sum of entries in row i)

Theory: Convergence of Pagerank

- given a matrix A, the spectral radius R(a) is defined as max | ev_i(A) | where ev_i(A) are the eigenvals of A
- Perron's Theorem: For a primitive matrix A:
  - there exists an eigenvalue R(a) with multiplicity 1
  - any other eigenvalue v has | v | < R(a)
- also, min_i r_i <= R(a) <= max_i r_i (where r_i is sum of entries in row i)
- Pagerank matrix: all rows add up to exactly 1
- thus, only one eigenvector for eigenvalue 1
- this is the unique solution to Pagerank

- why is it the eigenvector for eigenvalue 1?
- because no Pagerank disappears from the system!
- it only gets pushed from one node to another

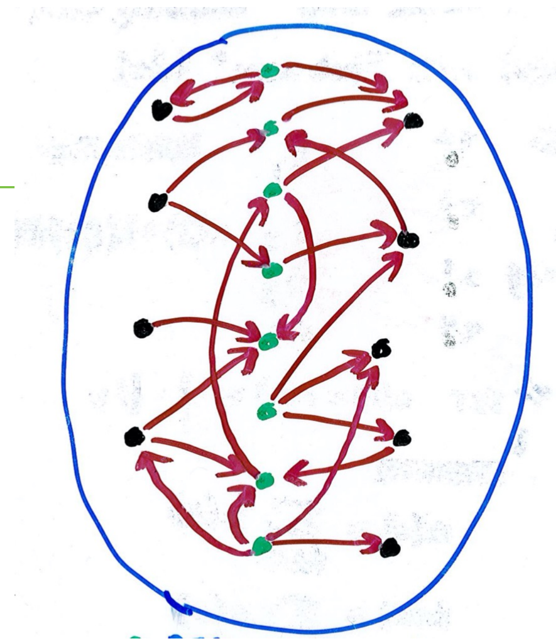
The HITS Algorithm (Kleinberg 1997)

- another algorithm using link structure for ranking
- but HITS is query-dependent, runs at query time
- Pagerank is query-independent, preprocessing step
- both HITS and Pagerank based on eigenvectors
- but different in interesting ways

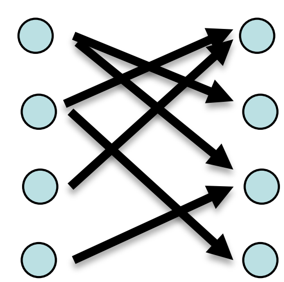
Outline of HITS

- run incoming query to retrieve top k results
- using any ranking algorithm, say BM25, for k = 200
- construct subgraph of the web consisting of:
  - the top k results
  - and pages linking to those top k results
  - and pages linked to by those top k results
- convert this subgraph into a bipartite graph
- run an iterative algorithm on bipartite graph

Outline of HITS

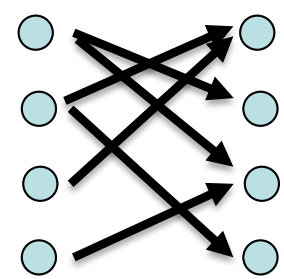


- top k results in green
- pages linking to those on left
- pages linked to by top k on right
- for each node in this subgraph, create two nodes
- one on left, one on right side of bipartite graph
- a link between node i on left and node j on right if node i in original graph links to node j



Hubs and Authorities

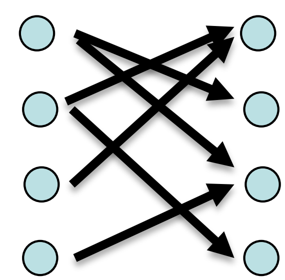
- an authority is a page with good info on a topic
- a hub is a page with good links for a topic
- a good hub points to good authorities
- a good authority is pointed at by good hubs
- each node is represented on each size of graph
- the left side nodes model hub quality
- those on right side model authoritativeness



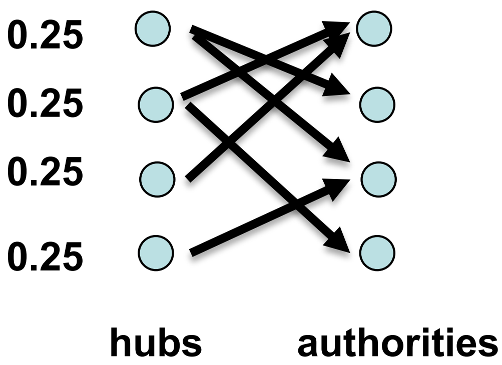
NYU | TANDON SCHOOL OF ENGINEERING

Iterative Algorithm for HITS

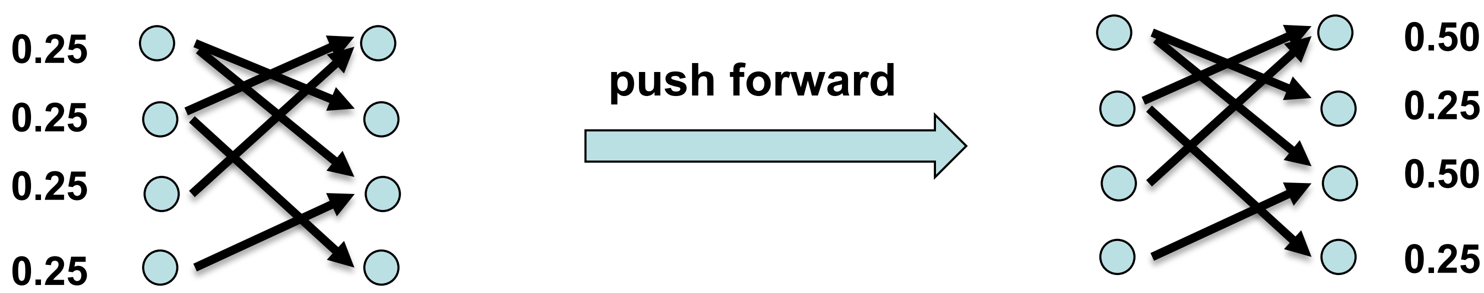
- one difference: we do not split score over edges
- initialize each node on the left with a score of 1/n
- do until convergence:
  - push score from left to right
  - push score from right to left
  - normalize the score vector



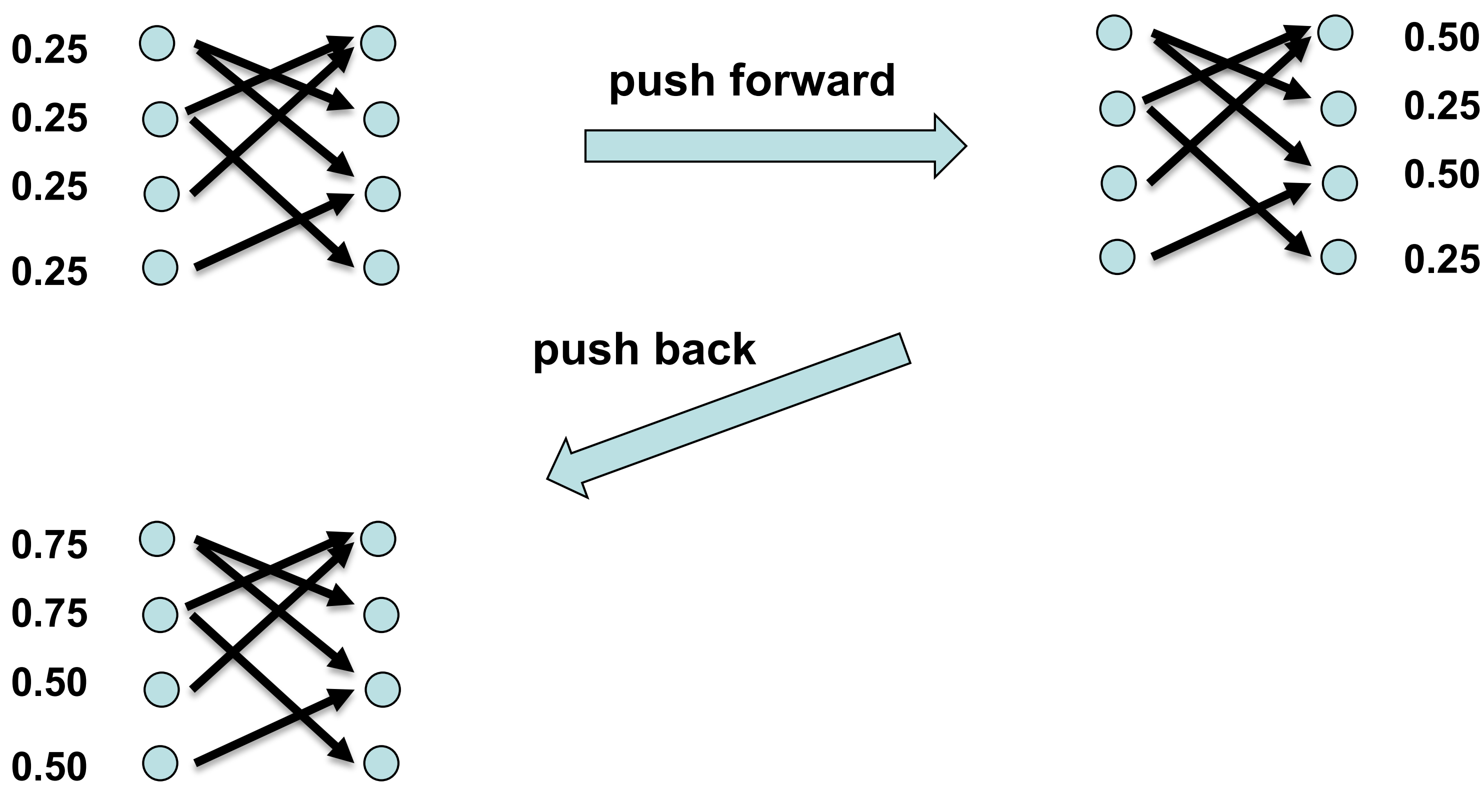
Iterative Algorithm for HITS



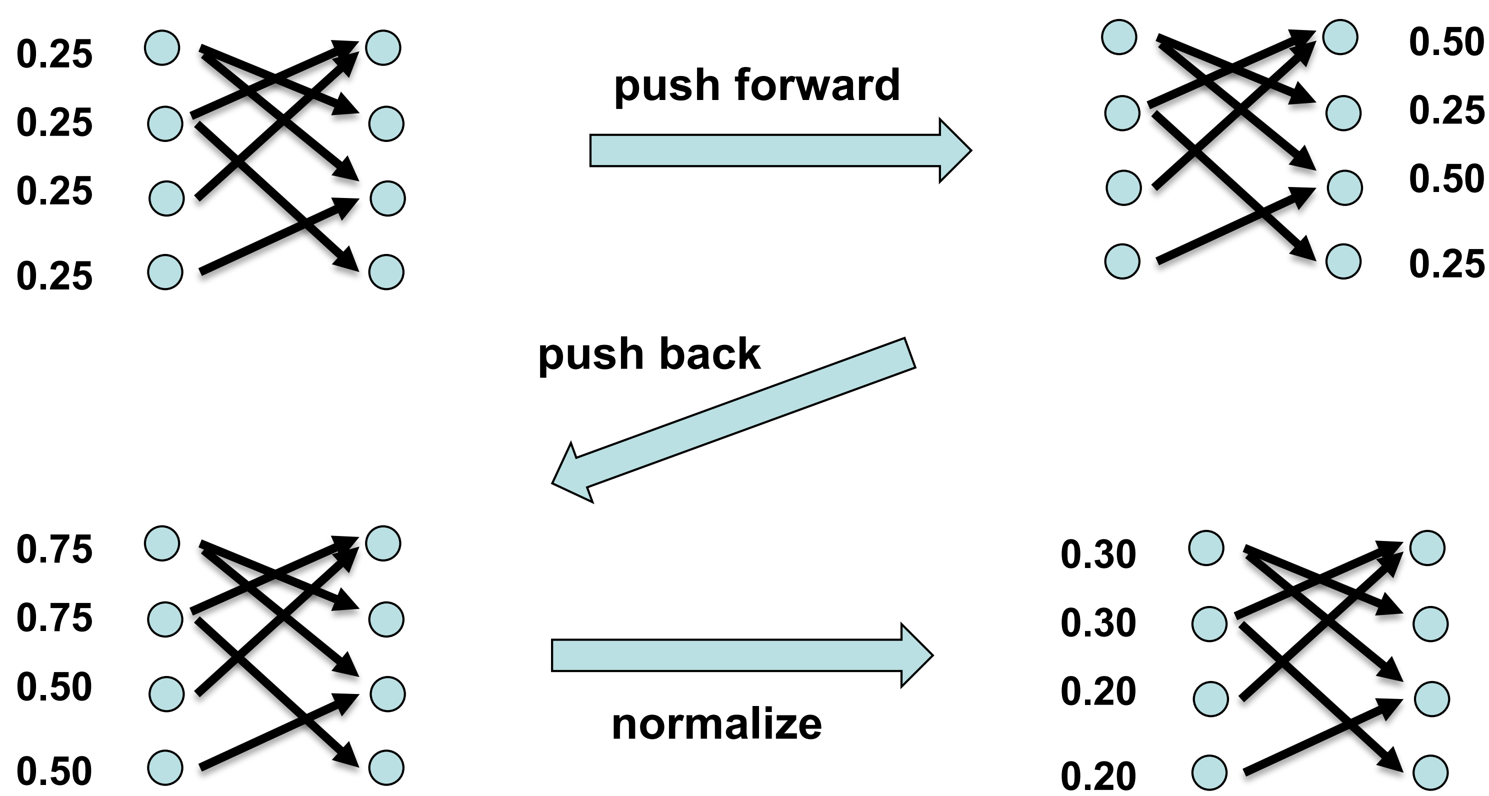
Iterative Algorithm for HITS



Iterative Algorithm for HITS



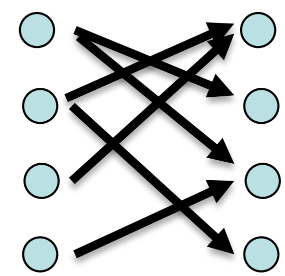
Iterative Algorithm for HITS



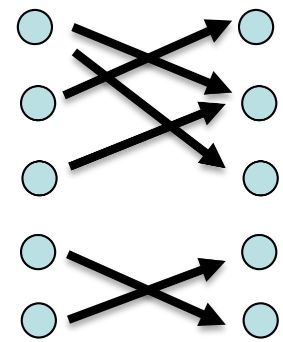
TANDON SCHOOL OF ENGINEERING

Convergence of HITS

- this always converges
- but solution may depend on initial assignment!
- eigenvector of A * A^T where A^T is transpose of A
- A is the forward matrix, A^T backward
- this is not always irreducible
- but different solutions mean things
- can figure out communities within results



Convergence of HITS



- two connected components, not irreducible
- initial assignment: all in first three, or all in last 2
- converges to different solutions
- but more complex in general due to normalization
- different solutions can highlight/expose different subcommunities

NYU TANDON SCHOOL OF ENGINEERING

Understanding the web graph is useful for

- Better ranking (Pagerank, HITS, etc.)
- Finding related pages, clusters, communities
- Focused crawling
- Better classification
- Spam detection
- ... and many other applications

- Challenge: how to efficiently compute with giant graphs
- E.g., 1 trillion nodes, 20 trillion edges

Main Research Directions on Web Graphs

- Efficient computing with very large graphs
  - Compressed graph structures for main memory
  - Parallel computing with large graphs: Pregel, graph DBs
  - I/O-efficient graph algorithms
- Basic web graph properties
  - Statistics, connectivity diameter
  - Random graph models for the web
- Graph Mining Applications
  - Link-based ranking
  - Classification
  - Related pages and communities

Compressed Graph Representations

- Goal: fit graph in memory (or most of it)
- DEC Connectivity Server (1998)
  - Graph data structure for answering link queries
  - Who does page x link to? And who links to x?
  - Was used in altavista search engine
  - Uncompressed (1998), later compressed (2002)
  - Can be used for many graph algos (e.g., HITS)
- A lot of subsequent work
- Using just a few bits per edge (3-6 bits)

Background: Graph Representations

- Adjacency matrix vs. adjacency list
- Adjacency matrix:
  - n x n matrix where n is number of nodes
  - A[i,j] = 1 iff there is an edge from node i to node j
  - Very sparse graph → wastes a lot of space
- Adjacency list:
  - For each node x, a list of other nodes x links to
  - Sparse representation of adjacency matrix
  - But needs another copy of edges for reverse links
- Most systems use adjacency lists

Background: Graph Representations

- **Adjacency matrix:**

  (5 nodes numbered 0 to 4)

to

from					
	0	1	0	1	1
	1	0	1	0	1
	0	1	0	1	0
	1	0	0	0	0
	0	1	1	1	0

- **Adjacency list:**

0	3	6	8	9	**list offsets**

1	3	4	0	2	4	1	3	0	1	2	3	**out-going neighbors**

Background: Graph Representations

- **Adjacency matrix:**

  (5 nodes numbered 0 to 4)

to

from					
	0	1	0	1	1
	1	0	1	0	1
	0	1	0	1	0
	1	0	0	0	0
	0	1	1	1	0

- **Adjacency list:**

0	3	6	8	9	**list offsets**

1	3	4	0	2	4	1	3	0	1	2	3	**out-going neighbors**

list of out-neighbors of node 0, starting at offset 0

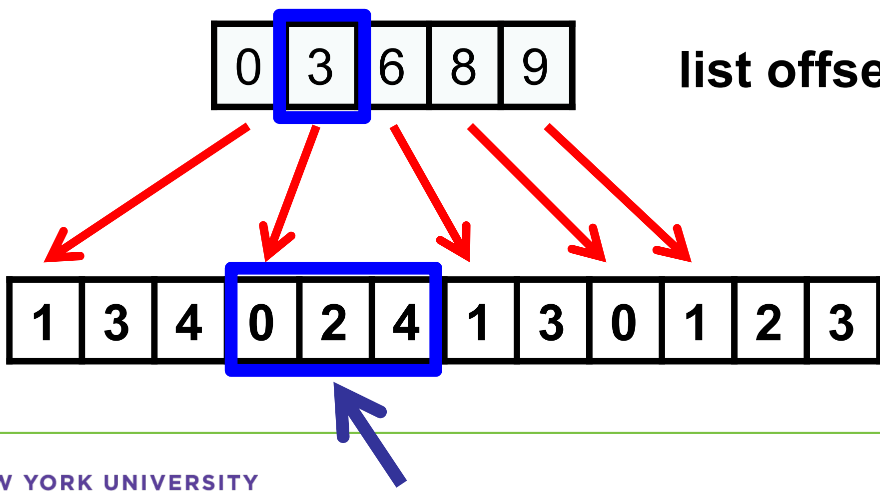
Background: Graph Representations

- **Adjacency matrix:**

  (5 nodes numbered 0 to 4)

	to				
from	0	1	0	1	1
	1	0	1	0	1
	0	1	0	1	0
	1	0	0	0	0
	0	1	1	1	0

- **Adjacency list:**



list offsets

out-going neighbors

list of out-neighbors of node 1, starting at offset 3

Background: Graph Representations

- **Adjacency matrix:**

  (5 nodes numbered 0 to 4)

to

from	0	1	0	1	1
	1	0	1	0	1
	0	1	0	1	0
	1	0	0	0	0
	0	1	1	1	0

- **Adjacency list:**

0	3	$\boxed{6}$	8	9	**list offsets**

1	3	4	0	2	4	$\boxed{1}$	$\boxed{3}$	0	1	2	3	**out-going neighbors**

Background: Graph Representations

- **Adjacency matrix:**

  (5 nodes numbered 0 to 4)

	to				
from	0	1	0	1	1
	1	0	1	0	1
	0	1	0	1	0
	1	0	0	0	0
	0	1	1	1	0

- **Adjacency list:**

list offsets

0	3	6	8	9

out-going neighbors

1	3	4	0	2	4	1	3	0	1	2	3

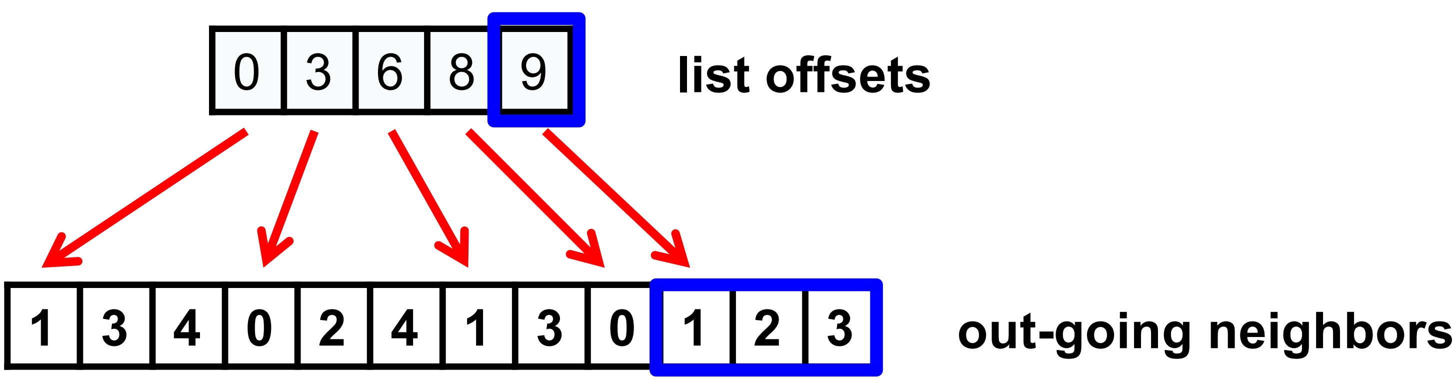
Background: Graph Representations

- **Adjacency matrix:**

  (5 nodes numbered 0 to 4)

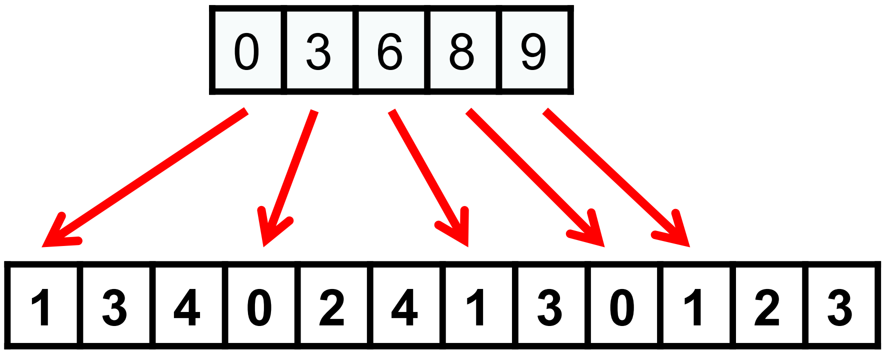
	to				
	0	1	0	1	1
	1	0	1	0	1
from	0	1	0	1	0
	1	0	0	0	0
	0	1	1	1	0

- **Adjacency list:**



Problem: Compressing Adjacency Lists

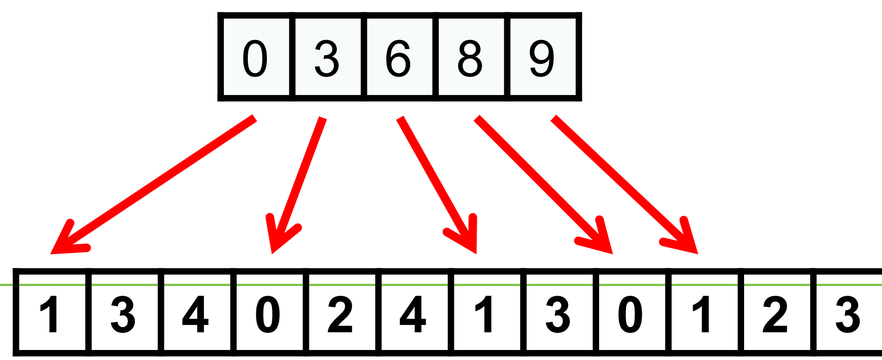
- Example:



- Similar to inverted lists
- But there is additional structure in graphs
- May need to compress edges in both directions

Main Ideas in Graph Compression

- Lists are sorted: take differences
- Many links to certain pages (use fewer bits for link to google.com)
- Most links (90%+) are local (go to other pages on same site)
- Links repeat between close-by pages (in URL ordering)
- Sets of links repeat as well (e.g., site menus)

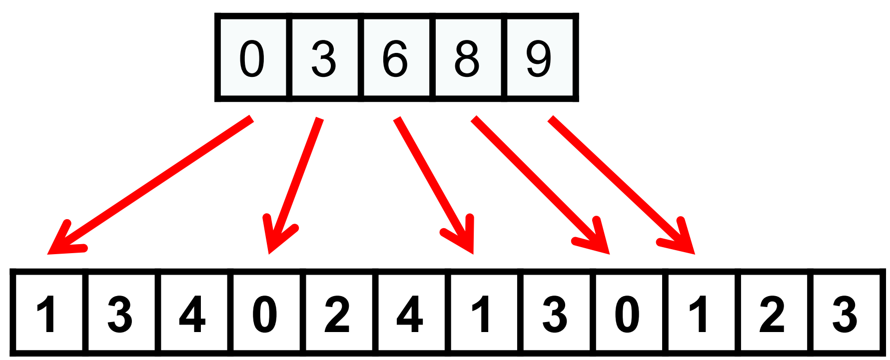


Summary: Graph Compression

- There is a large literature on compressing graphs
- Web graphs, social graphs, road maps, others
- Significant compression possible for many graphs
- Underlying ideas often simple
- Coding, taking differences, repetition of certain structures, locality, graph ordering and traversals
- Down to a few bits per edge for web graphs

Graph Structure Interface

- get_node_id (url)
- get_url (node_id)
- get_out-neighbors (node_id)
- get_in_neighbors (node_id)
- Enough for most graph algos



Parallel Computing with Large Graphs

- Use main memory (and CPU) on many nodes
- Many parallel graph algorithms
- Google Pregel (2010)
  - Graph partitioned over many nodes
  - Computation in phases (supersteps)
  - Nodes and edges have states
  - Nodes can send data over outgoing edges
  - ... on same of another machine
  - Based on Valiant’s Bulk-Synchronous Parallel model
- Open-source version: Apache Giraph
- Alternative: graph DBs such as neo4J

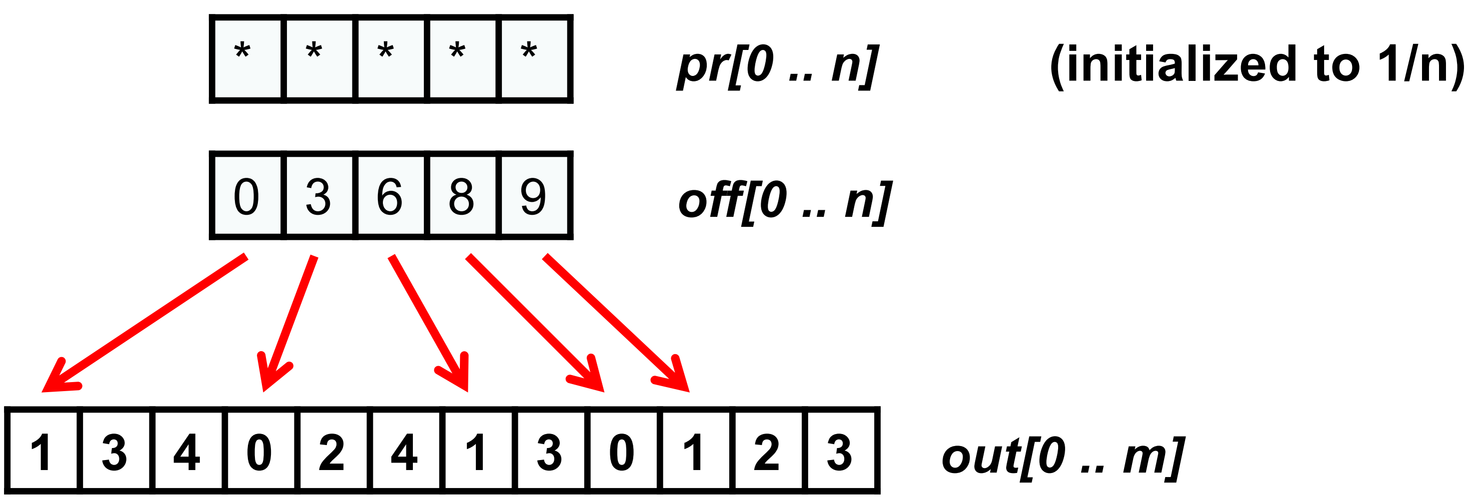
I/O-Efficient Graph Computing

- Suppose data does not fit in main memory
- Try to solve problems with mostly sequential I/O
- Ideally, a few iterations (scans etc) over the data
- Example: I/O-efficient sorting
- Common primitives: scan, merge, split, sort
- Many graph problems can be solved using sorting
- We can “communicate over edges” by sorting

- ... e.g., Pagerank

Pagerank: Setup

- Graph with n nodes and m edges
- Array *float pr[0 .. n]* of current pagerank values
- Array *int off[0 .. n]* of offsets of adjacency lists
- Array *int out[0 .. m]* of out-going neighbors



Computing One Iteration of Pagerank

- In pseudocode, with new pagerank in *newpr[0 .. n]*

```
for i = 0 to n-1
    newpr[i] = (1-beta)/n;            /* 1-beta is random jump prob */
for i = 0 to n-1
    outdegree = off[i+1] – off[i];    /* compute out-degree */
    for j = off[i] to off[i+1]-1;     /* iterate over out-edges */
        Newpr[out[j]] += beta * pr[i] / outdegree;
```

Computing One Iteration of Pagerank

- In pseudocode, with new pagerank in *newpr[0 .. n]*

```
for i = 0 to n-1
    newpr[i] = (1-beta)/n;              /* 1-beta is random jump prob */
for i = 0 to n-1
    outdegree = off[i+1] – off[i];      /* compute out-degree */
    for j = off[i] to off[i+1]-1;       /* iterate over out-edges */
        Newpr[out[j]] += beta * pr[i] / outdegree;
```

- Note: except for *newpr[0 .. n],* all arrays are only accessed sequentially

I/O-Efficient Pagerank

- Case 1: we can store on float for each node
  - Store pr[ ], off[ ], and out[ ] as files on disk
  - Store newpr[ ] in main memory
  - In each iteration, scan data from disk
  - Then write newpr[ ] to disk to replace pr[ ]

I/O-Efficient Pagerank

- Case 1: we can store on float for each node
  - Store pr[ ], off[ ], and out[ ] as files on disk
  - Store newpr[ ] in main memory
  - In each iteration, scan data from disk
  - Then write newpr[ ] to disk to replace pr[ ]
- Case 2: not enough memory for one float/node

I/O-Efficient Pagerank

- Case 1: we can store on float for each node
  - Store pr[ ], off[ ], and out[ ] as files on disk
  - Store newpr[ ] in main memory
  - In each iteration, scan data from disk
  - Then write newpr[ ] to disk to replace pr[ ]
- Case 2: not enough memory for one float/node
  - Instead of directly writing into newpr[ ]:

    newpr[out[j]] += beta * pr[i] / outdegree;
  - Create a record (out[j], beta * pr[i] / outdegree)
  - Basically, a note saying "*add this much pr to node out[j]*"
  - Write records out and sort them

I/O-Efficient Pagerank: Role of Sorting

- Sorting allows us to transmit data over edges
- Or, more generally, to read and write to random memory locations
- Every node creates record *read(i)* or *write(i)* where *i* is the memory location to read or write
- A number of reads (writes) reduce to one (two) sequential scans of memory (a file on disk)
- Efficient if enough parallelism and limited memory
- Basically, we can simulate PRAM computation

Pagerank in SQL

- Table pr(int nodeid, int outd, float pr)
- Table edges(int from, int to)

Pagerank in SQL

- Table pr(int nodeid, int outd, float pr)
- Table edges(int from, int to)

```
Insert into table newpr

Select edges.to, (1-beta)/n + beta*sum(pr.pr/pr.outd)
From pr, edges
Where pr.nodeid = edges.from
Group-by edges.to
```

Pagerank in SQL

- Table pr(int nodeid, int outd, float pr)
- Table edges(int from, int to)

```
Insert into table newpr

Select edges.to, (1-beta)/n + beta*sum(pr.pr/pr.outd)
From pr, edges
Where pr.nodeid = edges.from
Group-by edges.to
```

- Is this efficient?