DD2476: Search Engines and Information Retrieval Systems

Johan Boye*
KTH

Lecture 5

* Many slides inspired by Manning, Raghavan and Schütze

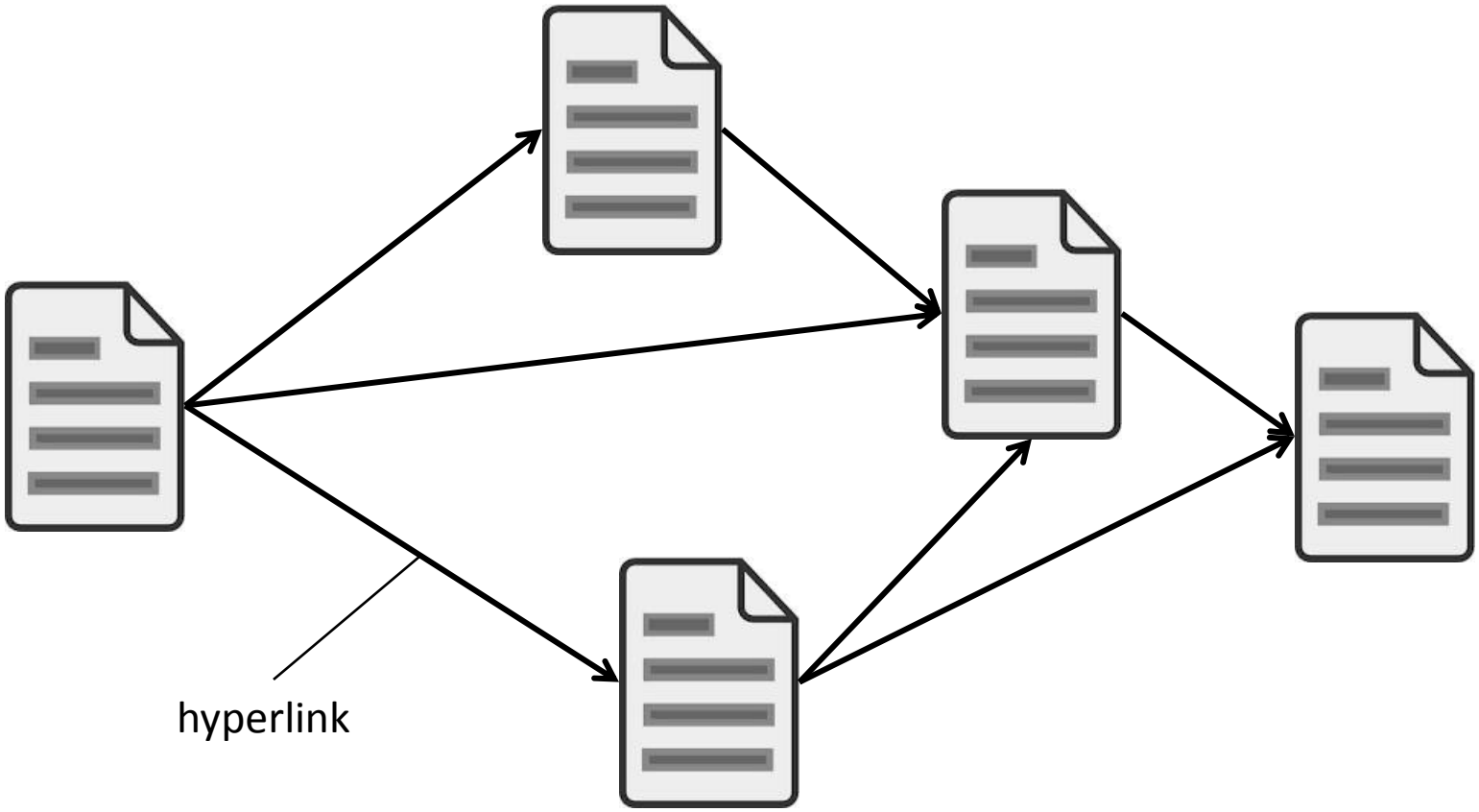
Static quality scores

- We want top-ranking documents to be both **relevant** and **authoritative**
  - Relevance – cosine scores
  - Authority – query-independent property

- Assign **query-independent quality score** g(*d*) in [0,1] to each document *d*

- net-score(*q*,*d*) = $w_1 \times g(d) + w_2 \times \cos(q,d)$

The Web as a directed graph



Using link structure for ranking

- **Assumption:** A link from X to Y signals that X's author perceives Y to be an authoritative page.
  - X "casts a vote" on Y.
- **Simple suggestion:** Rank = number of in-links
- However, there are problems with this naive approach.

PageRank: basic ideas

- WWW's particular structure can be exploited
  - pages have links to one another
  - the more in-links, the higher rank
  - in-links from pages having **high rank** are **worth more** than links from pages having low rank
  - this idea is the cornerstone of **PageRank** (Brin & Page 1998)

The random surfer model

- Imagine a **random surfer** that follow links
- The link to follow is selected with uniform probability
- If the surfer reaches a **sink** (a page without links), he **randomly restarts** on a new page
- Every once in a while, the surfer **jumps to a random page** (even if there are links to follow)

PageRank

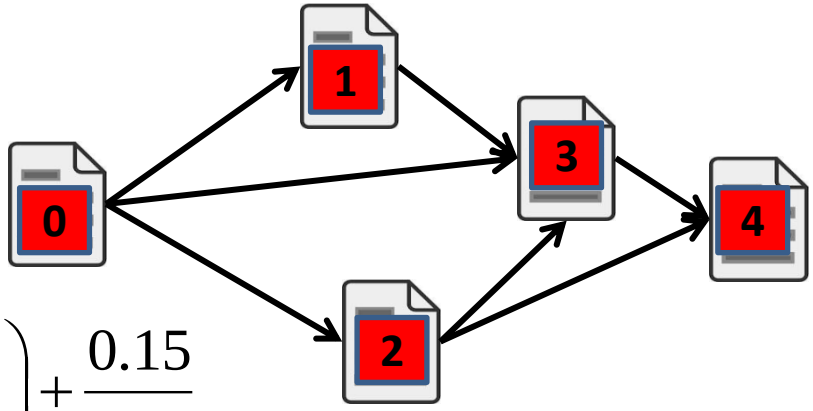
- With **probability *1-c*** the surfer is bored, **stops following links**, and **restarts** on a random page
- Guess: Google uses *c*=0.85

$$PR(p) = c\left(\sum_{q \in in(p)} \frac{PR(q)}{L_q}\right) + \frac{(1-c)}{N}$$

- Without this assumption, the surfer will get stuck in a subset of the web.

PageRank example

$$PR_4 = 0.85 \cdot \left(\frac{PR_2}{2} + PR_3 \right) + \frac{0.15}{5}$$



$$PR_3 = 0.85 \cdot \left(\frac{PR_0}{3} + PR_1 + \frac{PR_2}{2} + \frac{PR_4}{5} \right) + \frac{0.15}{5}$$

$$PR_2 = PR_1 = 0.85 \cdot \left(\frac{PR_0}{3} + \frac{PR_4}{5} \right) + \frac{0.15}{5}$$

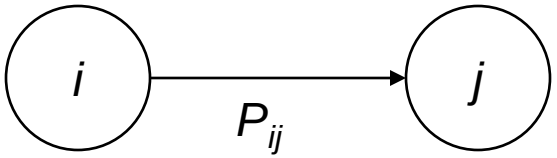
$$PR_0 = 0.85 \cdot \left(\frac{PR_4}{5} \right) + \frac{0.15}{5}$$

PageRank- interpretations

- Authority / popularity / relative information value
- PR_p = the probability that the random surfer will be at page *p* at any given point in time
- This is called the **stationary probability**
- How do we compute it?

Random surfer as a Markov chain

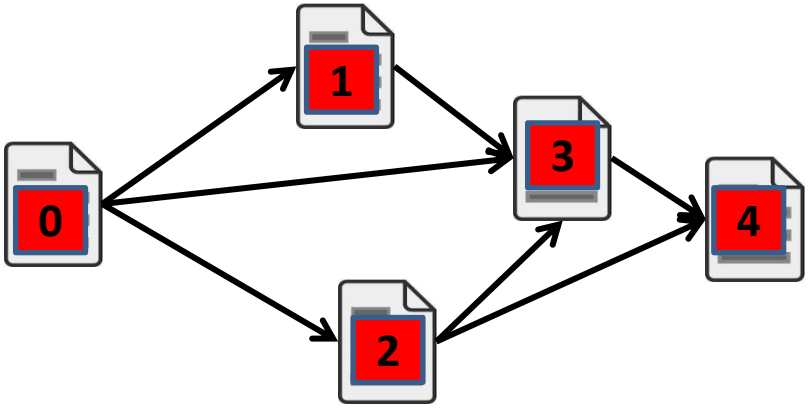
- The random surfer model suggests a a **Markov chain** formulation
  - A Markov chain consists of *n* states, plus an *n*×*n* transition probability matrix **P**.
  - At each step, we are in exactly one of the states.
  - For $1 \leq i,j \leq n$, the matrix entry P_{ij} tells us the probability of *j* being the next state, given we are currently in state *i*.



Transition matrices

P

0	0.33	0.33	0.33	0
0	0	0	1	0
0	0	0	0.5	0.5
0	0	0	0	1
0.2	0.2	0.2	0.2	0.2



J

0.2	0.2	0.2	0.2	0.2
0.2	0.2	0.2	0.2	0.2
0.2	0.2	0.2	0.2	0.2
0.2	0.2	0.2	0.2	0.2
0.2	0.2	0.2	0.2	0.2

G = cP+(1-c)J

0.0300	0.3105	0.3105	0.3105	0.0300
0.0300	0.0300	0.0300	0.8800	0.0300
0.0300	0.0300	0.0300	0.4550	0.4550
0.0300	0.0300	0.0300	0.0300	0.8800
0.2000	0.2000	0.2000	0.2000	0.2000

Ergodic Markov chains

- A Markov chain is **ergodic** if
  - you have a path from any state to any other
  - For any start state, after a finite transient time T_0, **the probability of being in any state at a fixed time $T>T_0$ is nonzero.**
- Our transition matrix is non-zero positive everywhere $\leftrightarrow$

  the graph is strongly connected $\leftrightarrow$

  the Markov chain is ergodic $\leftrightarrow$

  unique stationary probabilities exist

Probability vectors

- A **probability** (row) **vector x** = (x_1, … x_n) tells us where the walk is at any point.
- E.g., (000…1…000) means we're in state *i*.

  1 *i* *n*

- More generally, the vector **x** = (x_1, … x_n) means the walk is in state *i* with probability x_i.
- **xG** gives the next time step.
- **x** is stationary if **x=xG**

$$\sum_{i=1}^{n} x_i = 1.$$

Stationary probabilites

- Let **a** = (a_1, … a_n) denote the row vector of stationary probabilities.
  - If our current position is described by **a**, then the next step is distributed as **aG**.
  - But **a** is the steady state, so **a**=**aG**.
- Solving the matrix equation **x=xG** gives us **a**.
  - So **a** is the (left) eigenvector for **G**.
- **x** can be computed using **power iteration.**

Power iteration

- Start with any distribution (say **x**=(*10…0*)).
  - After one step, we're at **xG**;
  - after two steps at (**xG)G** , then **((xG)G)G** and so on.
- "Eventually", for "large" k, $\mathbf{xG}^k = \mathbf{a}$.

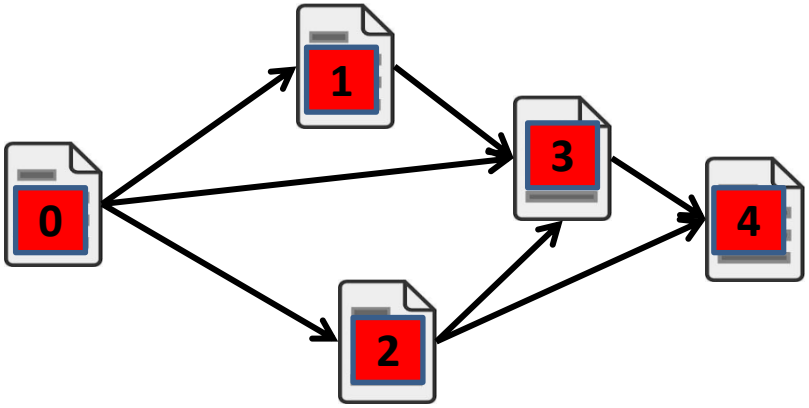
Power iteration algorithm

```
Let x=(0,…,0) and x′ an initial
  state, say (1,0,…,0)

while ( |x-x′| > ε ):
     x = x′
     x′= xG
```

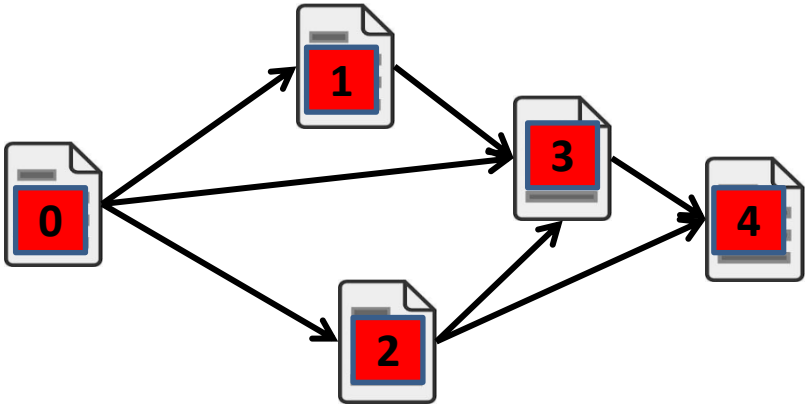
This algorithm converges **very** slowly.

Example



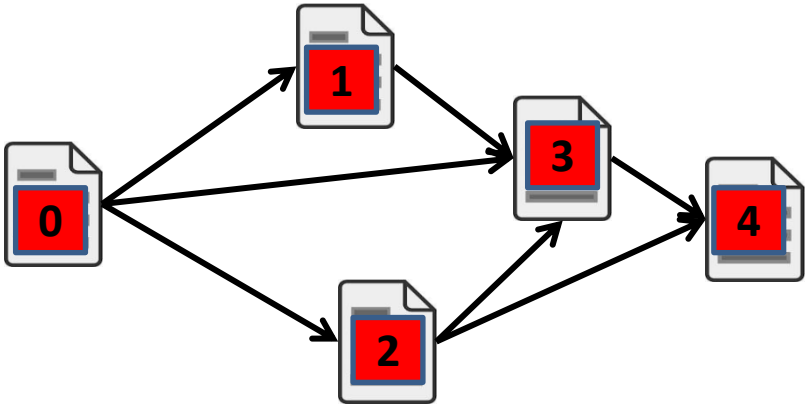
	t=0	t=1
0	1	.030
1	0	.313
2	0	.313
3	0	.313
4	0	.030

Example



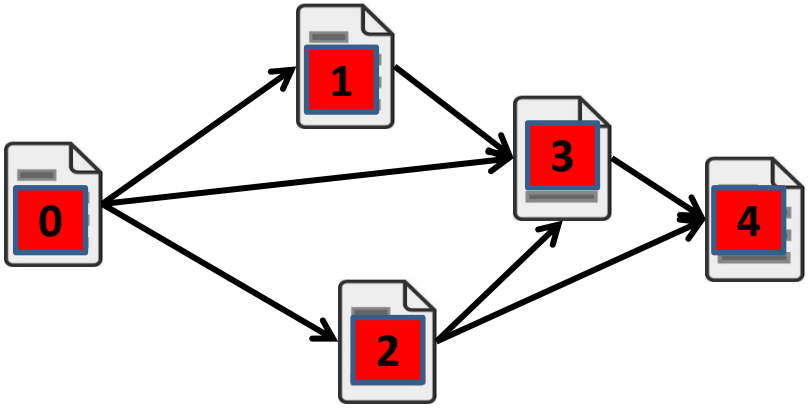
	t=0	t=1	t=2
0	1	.030	.035
1	0	.313	.044
2	0	.313	.044
3	0	.313	.443
4	0	.030	.435

Example



	t=0	t=1	t=2	t=3	t=4	t=5		t=16
0	1	.030	.035	.104	.115	.082	…	.095
1	0	.313	.044	.114	.144	.115	…	.122
2	0	.313	.044	.114	.144	.115	…	.122
3	0	.313	.443	.169	.289	.299	…	.278
4	0	.030	.435	.499	.307	.390	…	.383

Example



- Stationary probabilities:
  (0.095, 0.122, 0.122, 0.278, 0.383)

  found after 16 iterations starting from
  (1, 0, 0, 0, 0)

Example

- Stationary probabilities:
  x = (0.095, 0.122, 0.122, 0.278, 0.383)
- Check: xG =

$$(0.095 \quad 0.122 \quad 0.122 \quad 0.278 \quad 0.383) \begin{bmatrix} 0.0300 & 0.3105 & 0.3105 & 0.3105 & 0.0300 \\ 0.0300 & 0.0300 & 0.0300 & 0.8800 & 0.0300 \\ 0.0300 & 0.0300 & 0.0300 & 0.4550 & 0.4550 \\ 0.0300 & 0.0300 & 0.0300 & 0.0300 & 0.8800 \\ 0.2000 & 0.2000 & 0.2000 & 0.2000 & 0.2000 \end{bmatrix}$$

$$= (0.0951 \quad 0.1218 \quad 0.1218 \quad 0.2773 \quad 0.3833)$$

- So x is (approximately) the left eigenvector of G.

Approximating PageRank

- For large graphs, power iteration takes a loooong time.
- We can also approximate PageRank by **simulating the random surfer**.

Monte Carlo simulation to approximate PageRank

- Avrachenkov et al. describe a number of simulation algorithm based on the random surfer model
- D = document id
- Consider a random walk $\{D_t\}_{t\geq 0}$ that starts from some page.
- At each step t:
  - Prob c: D_t = one of the documents with edges from $D_{t\text{-}1}$
  - Prob $(1 - c)$: The random walk terminates

1. MC end-point with random start

- Simulate N runs of the random walk $\{D_t\}_{t\geq 0}$ initiated at a **randomly chosen page**
- PageRank of page j = 1,...,n:

  $\boldsymbol{\pi_j}$ **= (#walks which end at j)/N**
- **Worst case: N = O(n^2)**
- **Mean case: N = O(n)**

2. MC end-point with cyclic start

- Simulate N = mn runs of the random walk $\{D_t\}_{t\geq 0}$ initiated at **each page exactly m times**
- PageRank of page j = 1,...,n:

  π_j = (#walks which end at j)/N

4. MC complete path stopping at dangling nodes

- Simulate N = mn runs of the random walk $\{D_t\}_{t\geq 0}$ initiated at **each page exactly m times** and **stopping when it reaches a dangling node**
- PageRank of page j = 1,...,n:

  π_j = (#visits to node j during walks)/
  (total #visits during walks)

5. MC complete path with random start

- Simulate N runs of the random walk $\{D_t\}_{t\geq 0}$ initiated at a **randomly chosen page** and **stopping when it reaches a dangling node**
- PageRank of page j = 1,...,n:

  π_j = (#visits to node j during walks)/(total #visits during walks)

Monte Carlo advantages

- **Power interation:**
  - precision improves linearly for all docs
  - computationally expensive
  - must be redone when new pages are added

- **Monte Carlo method:**
  - precision improves faster for high-rank docs
  - parallel implementation possible
  - continuous update

Assignment 2

- 2.5 Implement power iteration
  Compute pageranks of links1000.txt
- 2.6 Power iteration with sparse matrices
  Compute pageranks of linksDavis.txt
  Combine tf-idf and pagerank
- 2. 7 Implement Monte-Carlo pagerank computations
  Compute pageranks of linksWiki.txt
- 2.8 Hubs and authorities

HITS (Hypertext-Induced Topic Selection)

- HITS is an method similar to PageRank
  - proposed the same year (1998)
- Goal is to find high-quality pages of two kinds:
  - **Hubs**: linking to pages with great authority
  - **Authorities**: linked from great hubs

A typical hub

culture trip

Explore your world

Sections Wis

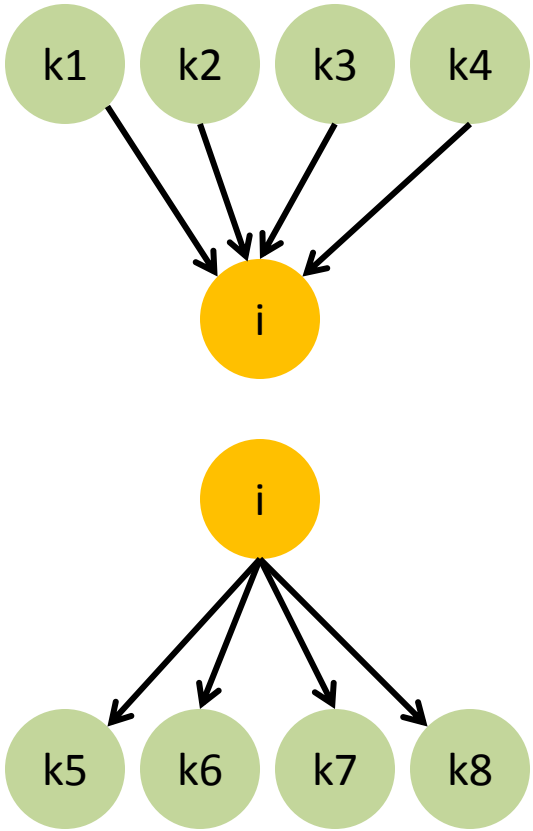
Home › Sweden › Stockholm ›

Stockholm / BARS & CAFES

The 10 Best Cafes And Coffeehouses In Stockholm

Hubs and authorities

- Each page *i* has two scores:
  - authority score a_i
  - hub score h_i
- HITS algorithm: Initialize $a_i = 1$ $h_i = 1$
- Then iterate until convergence:
  - Authority: $\forall i : a_i := \sum_{k \to i} h_k$
  - Hub: $\forall i : h_i := \sum_{i \to k} a_k$
  - Normalize $\sqrt{\sum_i a_i^2} := 1 \quad \sqrt{\sum_i h_i^2} := 1$



Matrix formulation

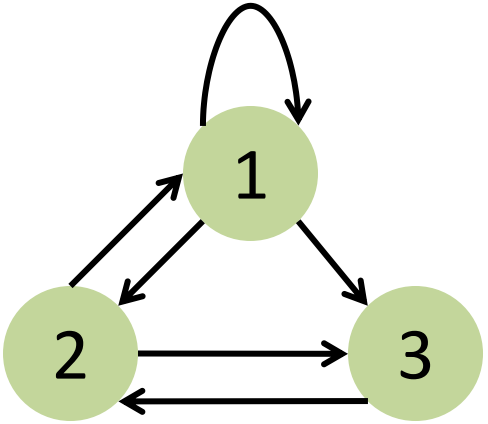
- $a = (a_1, \ldots, a_n)$ $h=(h_1,\ldots, h_n)$ are authority/hub scores for n documents
- A is the **adjacency matrix**: $A_{ik} = 1$ iff $i \rightarrow k$
- Then do:

$$h_i := \sum_{i \rightarrow k} a_k \Leftrightarrow h_i := \sum_k A_{ik} a_k$$

- i.e. $h := Aa$
- and likewise $a := A^T h$

HITS algorithm (example)

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

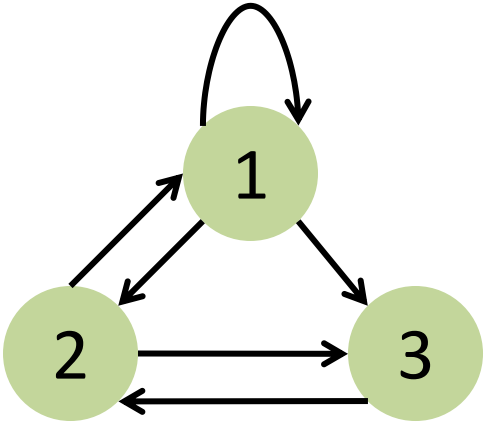


a(1)	1					
a(2)	1					
a(3)	1					

h(1)	1					
h(2)	1					
h(3)	1					

HITS algorithm (example)

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$



$a = A^T h$

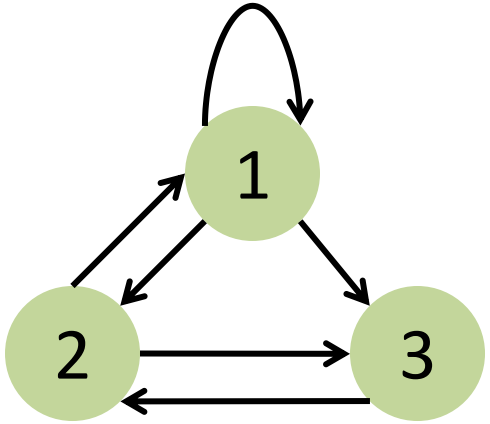
a(1)	1	2				
a(2)	1	2				
a(3)	1	2				

$h = Aa$

h(1)	1	3				
h(2)	1	2				
h(3)	1	1				

HITS algorithm (example)

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$



Normalize

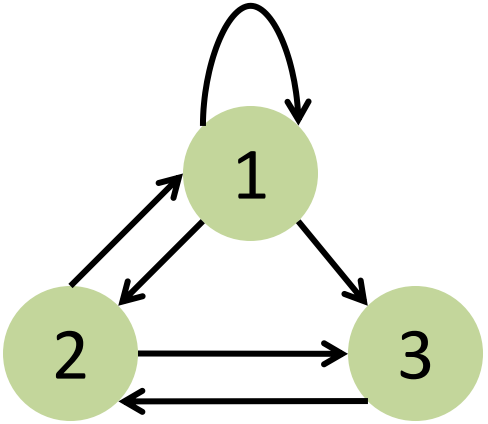
a(1)	1	2	2/√12=0.577		
a(2)	1	2	2/√12=0.577		
a(3)	1	2	2/√12=0.577		

Normalize

h(1)	1	3	3/√14 = 0.802		
h(2)	1	2	2/√14=0.535		
h(3)	1	1	1/√14=0.267		

HITS algorithm (example)

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$



a= A^Th, then normalize

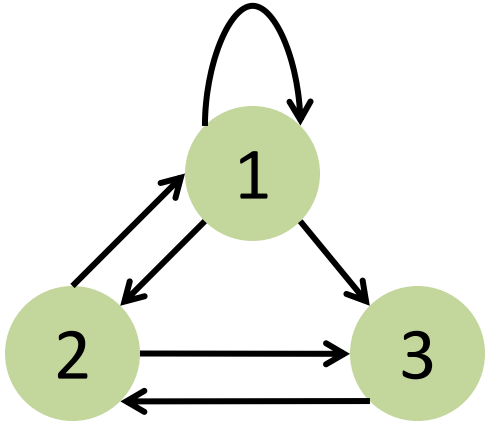
a(1)	1	0.577	0.615			
a(2)	1	0.577	0.492			
a(3)	1	0.577	0.615			

h= Aa, then normalize

h(1)	1	0.802	0.793			
h(2)	1	0.535	0.566			
h(3)	1	0.267	0.226			

HITS algorithm (example)

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$



a(1)	1	0.577	0.615	…	…	0.627
a(2)	1	0.577	0.492	…	…	0.460
a(3)	1	0.577	0.615	…	…	0.627

h(1)	1	0.802	0.793	…	…	0.789
h(2)	1	0.535	0.566	…	…	0.577
h(3)	1	0.267	0.226	…	…	0.211

Convergence

Results of the HITS algorithm

- When we have reached convergence, we have:

$h = c_h A a$ and $a = c_a A^T h$

where c_h and c_a are scalars.

i.e. $h = \lambda_h A A^T h$ and $a = \lambda_a A^T A a$

where h is the eigenvector of AA^T, and λ_h is the eigenvalue.
Likewise, a is the eigenvector of A^TA, and λ_a eigenvalue.

Results of the HITS algorithm

- Check:

$$AA^T h = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0.793 \\ 0.566 \\ 0.226 \end{bmatrix} = \begin{bmatrix} 3.732 \\ 2.729 \\ 0.998 \end{bmatrix} = 4.73 \begin{bmatrix} 0.793 \\ 0.566 \\ 0.226 \end{bmatrix}$$

i.e. h is an eigenvector of AA^T, with corresponding eigenvalue 4.73.

Using the HITS algorithm

- Unlike PageRank, the hubs and authorities scores are **computed at query time**
- Given a query q:
  - the **root set** are all documents that include at least one query word
  - the **base set** are all documents linking to, or linked from, a root doucument
- We then use the base set for computing the hubs and authorities scores

Tolerant retrieval

- Spelling correction
  - *"see you on the wki"* $\rightarrow$ *"see you on the wiki"*
- Wildcard queries
  - *"colo*rful"* $\rightarrow$ *"colorful"*, *"colourful"*
- In both cases, the search engine needs to
  - construct the intended query (or queries)
  - compute the results for those queries (intersection, phrase, ranked retrieval)
  - list the results

Wildcard queries: one word

- **`care*:`** find all docs containing any word beginning "care".
- **`*less:`** find words ending in "less"
- **`colo*r:`** find all words beginning "colo" and ending in "r"
- general case: any numbers of '*****' placed anywhere in the word (we will **not** consider this case)
- special case: '*****' matches all words (you don't need to consider this case)

Wildcard queries: several words

- **`b* colo*r:`** find all docs containing any word beginning “b”, **and** any word beginning with “color” ending in “r”

Wildcard queries

- How do we find all words matching **`care*`** ?
- **Idea:** Go through all words in the vocabulary, and check which words match the regular expression **`^care.*`**
  - e.g. using Java's regex library
- Would this work?

K-gram index

- For both wildcard queries and spelling correction we must **quickly** find words that
  - the user intended (for wildcard queries), or
  - the user probably intended (for spelling correction)

- A ***k*-gram index** is an index from k-grams (parts of words) to words.
  - **bigram** index when k=2
  - **trigram** index when k=3
  - etc.

K-grams

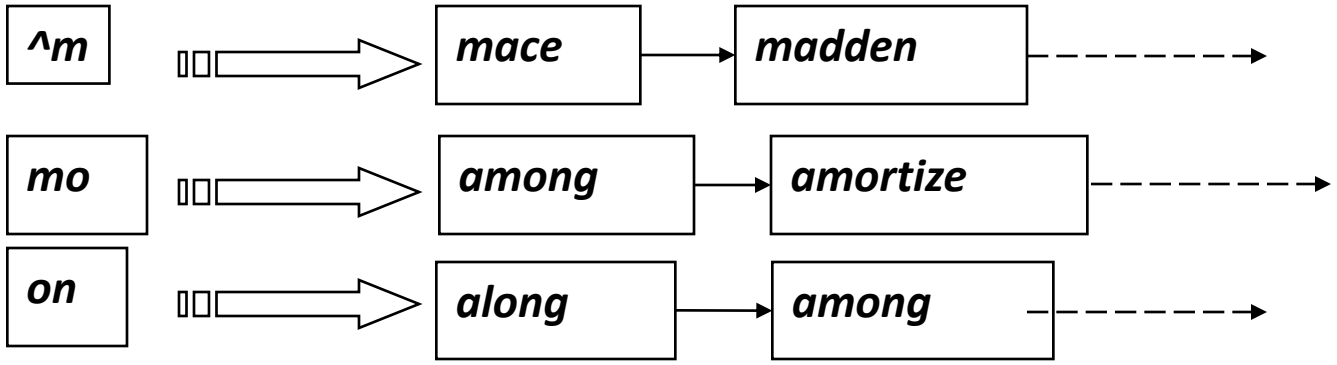
- The bigrams of **`december`**:
  - First add start and end symbol: **`^december$`**
  - Bigrams are all two-letter sequences:
    `^d, de, ec, ce, em, mb, be, er, r$`

- The trigrams of **`december`**:
  - **`^de, dec, ece, cem, emb, mbe, ber, er$`**

- A word of length *n* has *n+3–k* *k*-grams

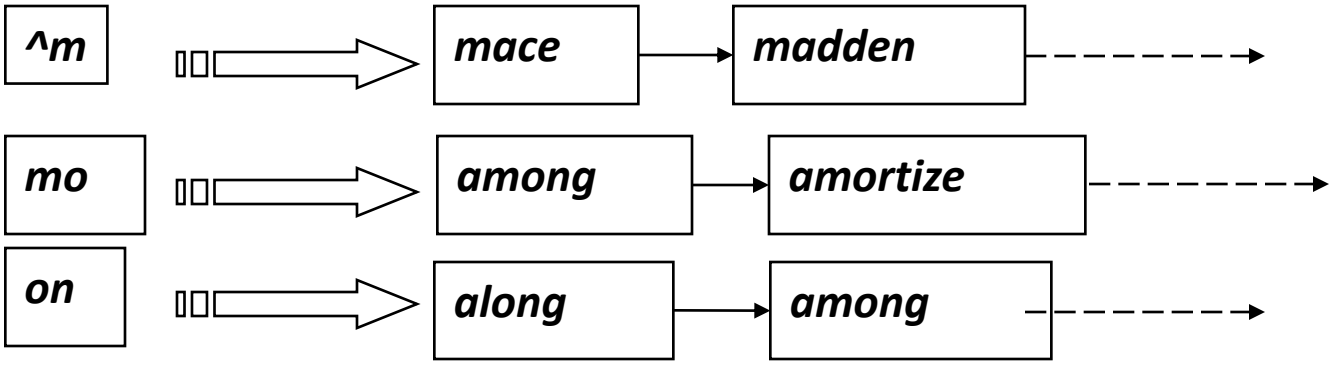
K-gram index

- For k-gram indexes, we can reuse a lot of ideas from our usual inverted indexes:
  - keys in a hashtable
  - values as arraylists



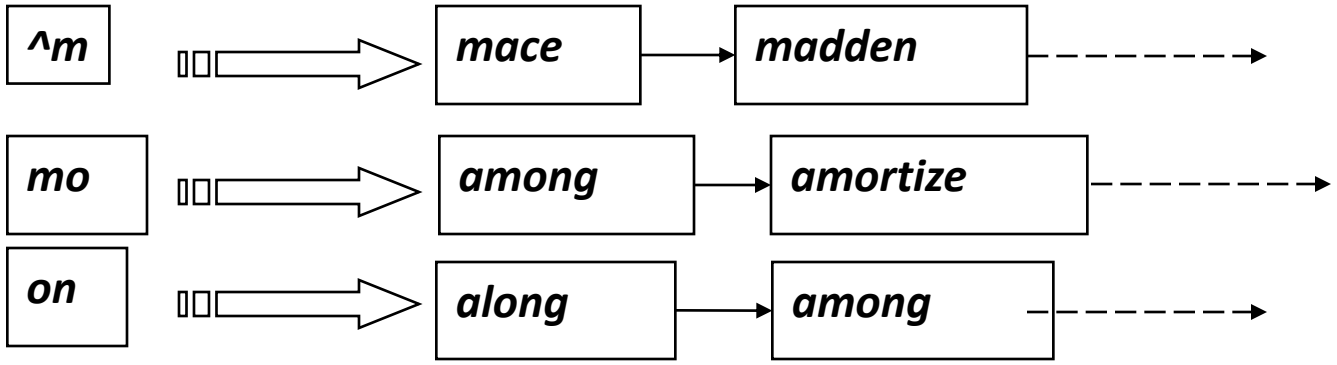
K-gram index and wildcards

- Suppose we want to find matches for **`mon*`**
- How would you search?



K-gram index and wildcards

- Suppose we want to find matches for **`mon*`**
- How would you search?
  - do an **intersection** search for **`^m mo on`** in k-gram index
  - post-process the results using the regex library
  - do a **union** search on the resulting words in the ordinary index

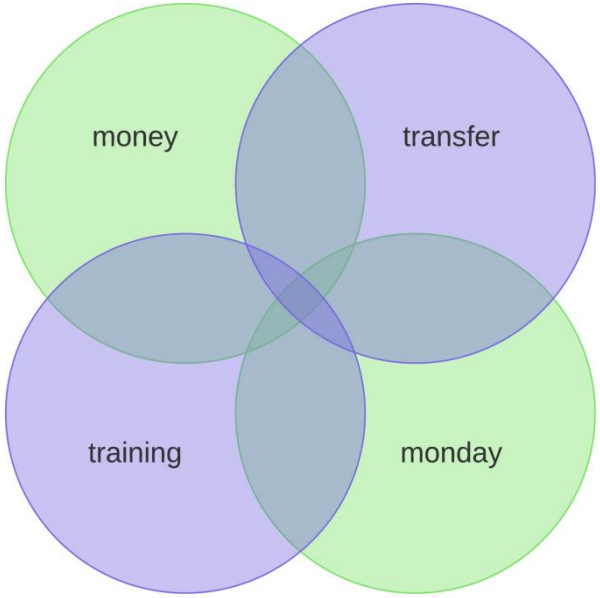


K-gram index and wildcards

- Suppose we want to find matches for

  `mon* tra*`

- Which documents in this Venn diagram are you looking for?



Spelling correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve “right” answers
    - Usually documents are left intact, but queries spell-checked
- Two main flavors:
  - Isolated word
    - Will not catch typos resulting in correctly spelled words, e.g., ***from → form***
  - Context-sensitive, e.g. ***I flew <u>form</u> Heathrow to Narita.***

Spelling correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - (**Grammatical approach**) A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - (**Data-driven approach**) The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

Spelling correction of a single word

- What we will do in assignment 3:
  - Data-driven approach (no lexicon)
  - Assumption: A word is **not** misspellt if it appears in at least 1 document.
  - If a word has 0 occurrences it **might** be misspellt, and the search engine should suggest corrections

Spelling correction of a single word

- Methods:
  - **Edit distance**
  - **Weighted edit distance**
  - ***n*-gram overlap**
- These can (should?) be combined

Levenshtein (edit) distance

- What is dist(*intention*, *execution*)?

```
i n t e n t i o n
n t e n t i o n
e t e n t i o n
e x e n t i o n
e x e n u t i o n
e x e c u t i o n
```

← delete *i*

← substitute *n* by *e*

← substitute *t* by *x*

← insert *u*

← substitute *n* by *c*

- Cost 1+2+2+1+2 = 8
- Can be efficiently computed with dynamic programming

Computing Levenshtein distance

e						
k						
o						
r						
b						
#						
	#	a	b	o	v	e

Computing Levenshtein distance

e	5					
k	4					
o	3					
r	2					
b	1					
#	0	1	2	3	4	5
	#	a	b	o	v	e

Computing Levenshtein distance

e	5					
k	4					
o	3					
r	2					
b	1	2				
#	0	1	2	3	4	5
	#	a	b	o	v	e

Computing Levenshtein distance

e	5					
k	4					
o	3					
r	2					
b	1	2	1			
#	0	1	2	3	4	5
	#	a	b	o	v	e

Computing Levenshtein distance

e	5	6	5	4	5	4
k	4	5	4	3	4	5
o	3	4	3	2	3	4
r	2	3	2	3	4	5
b	1	2	1	2	3	4
#	0	1	2	3	4	5
	#	a	b	o	v	e

Weighted edit distances

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors, e.g. ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Using edit distances

- Given query, enumerate all strings within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of "correct" words
  1. Show terms you found to user as suggestions, ***or***
  2. Look up all possible corrections in our inverted index and return all docs … slow, ***or***
  3. Run with a single most likely correction
- In assignment 3, we will opt for alternative 1.

Using edit distances

- Given a query, do we compute its edit distance to every dictionary term?
  - Expensive and slow
- How do we find the candidate dictionary terms?
  - One alternative: k-gram overlap
  - Use the k-gram index again!
  - Can also be used by itself for spelling correction

k-gram overlap

- Enumerate all the *k*-grams in the query string as well as in the lexicon
  - ***november:*** `^no, nov, ove, vem, emb, mbe, ber, er$`
  - ***december:*** `^de, dec, ece, cem, emb, mbe, ber, er$`
  - overlap 4/12 unique trigrams
  - the **Jacquard coefficient** = 4/12 = 0.33
  - generally, $\frac{|X \cap Y|}{|X \cup Y|}$ where X,Y are sets

Spelling correction of a single word

- E.g. **`wki`**
  - Do a union search in the k-gram index for **`^w wk ki i$`**
  - Calculate the Jacqard coefficient between **`wki`** and each of the resulting words
  - If the JC > some threshold for word *w*, calculate the Levenshtein distance between *w* and **`wki`**
  - If Levenstein distance < some other threshold, then *w* is a potential correction
  - Add *w* to the list of corrections

Spelling correction, multi-word queries

- E.g. "**`See yuoq on the wki`**"
- Includes two (possibly) misspellt words: **`yuoq`** and **`wki`** with 0 postings
- Construct the lists of spelling suggestions for each word
  - Lists for words with > 0 postings will only contain themselves
  - Then merge the lists

Spelling correction, multi-word queries

- **`See youq on the wki`**

see	you	on	the	wiki
	your			ki
	youd			wi
	yous			wk
	youn			waki
	…			…
	…			…
	…			…

- Now the lists have to merged to produce final suggestions

Spelling correction, multi-word queries

- Final list of suggestions (for instance):
  - **`See you on the wiki`**
  - **`See you on the ki`**
  - **`See you on the wi`**
  - **`See you on the wk`**
  - **`See you on the waki`**

General issues in spell correction

- We enumerate several possible alternatives to misspelled queries – which ones should we present to the user?
- Use heuristics:
  - The alternative matching most documents (expensive)
  - **The alternative likely to match most documents** (using heuristics, cheaper)
  - Query log analysis – what have others been searching for? What has this user been searching for?

Assignment 3

- 3.3 K-gram index
- 3.4 Wildcard queries
- 3.5 Spell checking, isolated queries
- 3.6 Spell checking, multi-word queries