Web Intelligence

Lecture 1: Intro and Crawling

Manfred Jaeger
with material from Peter Dolog and Bo Thiesson

AALBORG UNIVERSITET

Logistics

People

Lecturer: Manfred Jaeger `jaeger@cs.aau.dk`
Teaching Assistant: Abiram Mohanaraj `abiramm@cs.aau.dk`

Schedule

Exercises: Mondays 12:30-14:15 in the group rooms
Lectures: Mondays 14:30-16:15 NOVI 9
Extended Exercises/Self-study: Wednesday mornings in the group rooms

Exercises

Regular exercises: smaller problems solved with pencil and paper (or whiteboard, or . . .)

Self study exercises:

- larger practical programming tasks using Python and Jupyter notebooks
- closely aligned with exam topics
- exam option: answer exam questions using your self study exercise notebooks

Main source for the first part of the course:

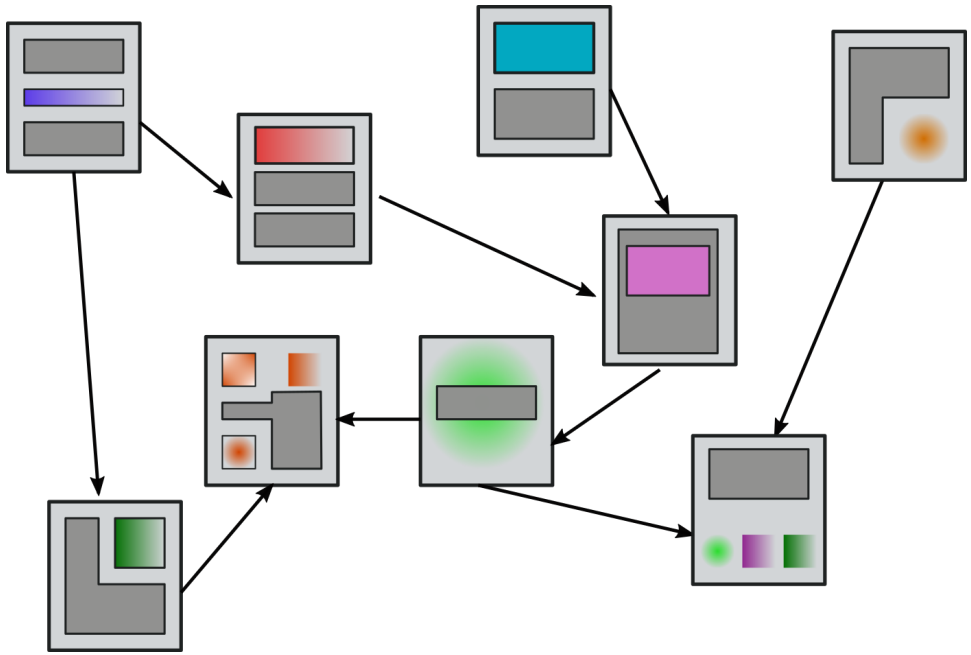
Introduction to Information Retrieval by Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze

Full text online:

`https://nlp.stanford.edu/IR-book/information-retrieval-book.html`

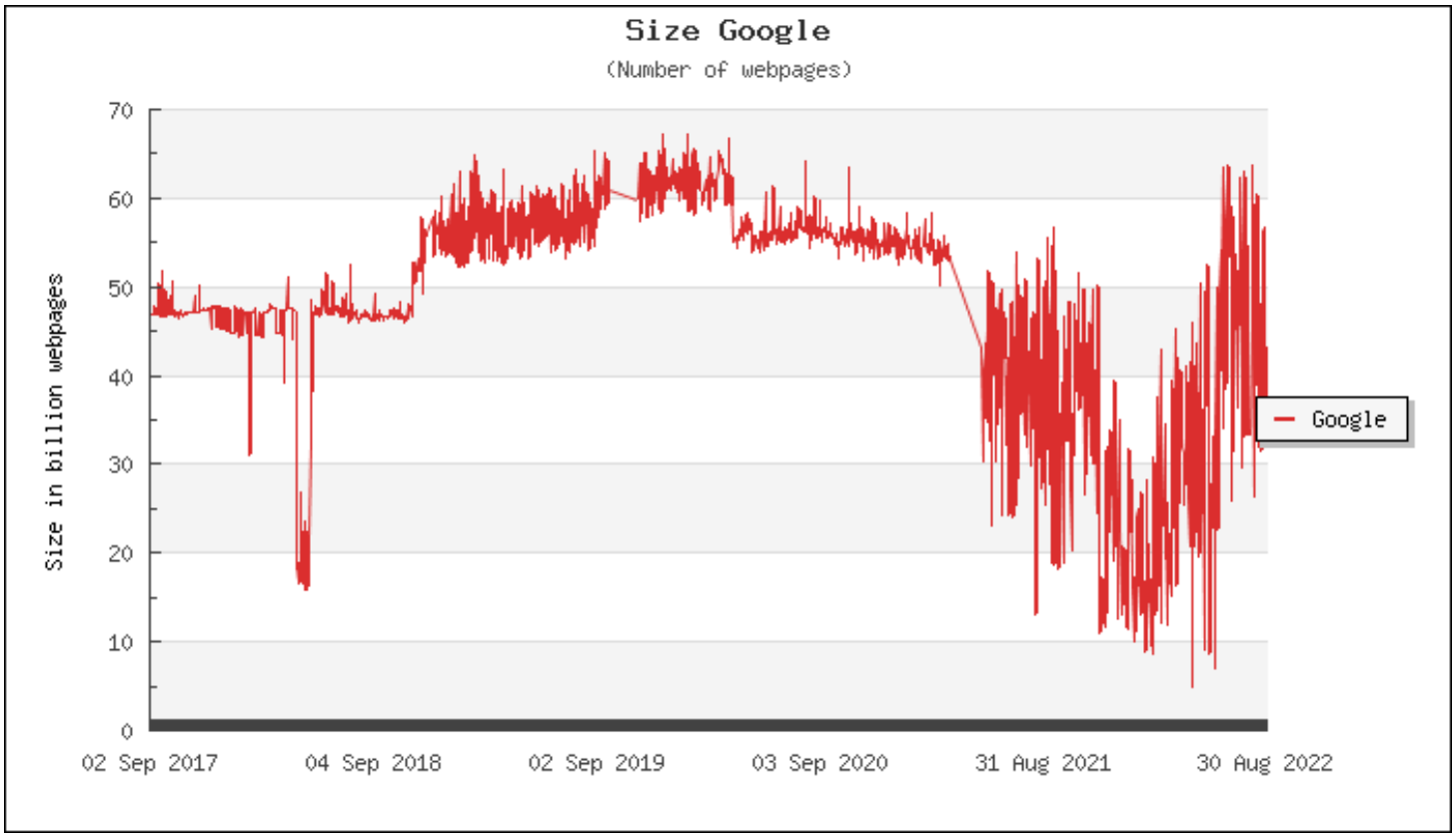
Later parts of the course will depend more on single book chapters and/or research articles.

Introduction



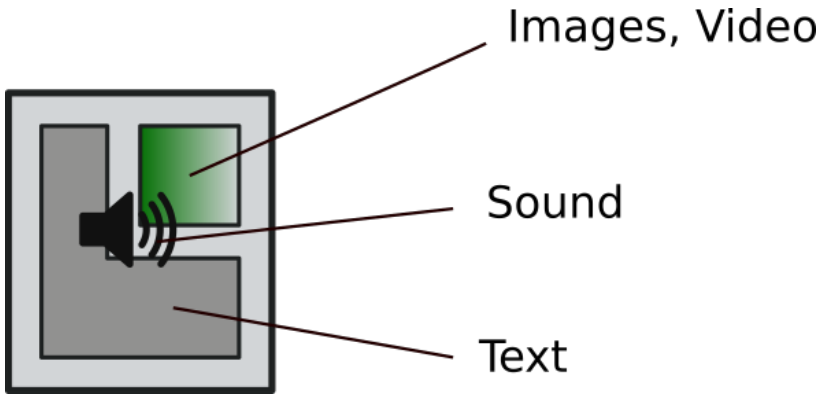
Hyper-linked network of web pages

Number of web-pages indexed by Google estimated by
`https://www.worldwidewebsize.com/`



World population: $\sim$ 8 billion.

Multi-media content:



We focus on text:

- Carries the most important information (in most cases)
- Techniques used for dealing with text can be adapted to dealing with other types of data (cf. “visual words” in computer vision).

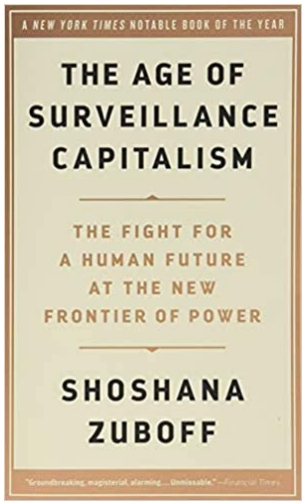
Intelligent ways to extract information and knowledge from the web:

- finding relevant information available on the web
- obtaining new knowledge by analyzing web data: the web itself, but also how it evolves, and how users interact on and with the web

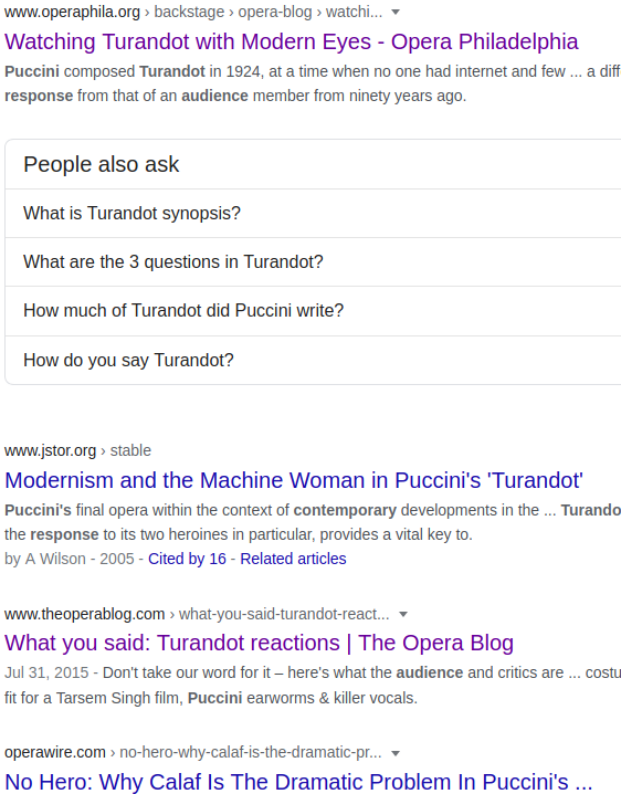
Some applications:

- Intelligent Search
- Recommender Systems
- Business Analytics
- Crowd Sourcing
- Not so nice ones: advertising, manipulation, surveillance

A critical view:



Query: Contemporary reaction Puccini Turandot



www.operaphila.org › backstage › opera-blog › watchi...

Watching Turandot with Modern Eyes - Opera Philadelphia

Puccini composed **Turandot** in 1924, at a time when no one had internet and few ... a diff **response** from that of an **audience** member from ninety years ago.

People also ask

What is Turandot synopsis?

What are the 3 questions in Turandot?

How much of Turandot did Puccini write?

How do you say Turandot?

www.jstor.org › stable

Modernism and the Machine Woman in Puccini's 'Turandot'

Puccini's final opera within the context of **contemporary** developments in the ... **Turando** the **response** to its two heroines in particular, provides a vital key to.

by A Wilson - 2005 - Cited by 16 - Related articles

www.theoperablog.com › what-you-said-turandot-react...

What you said: Turandot reactions | The Opera Blog

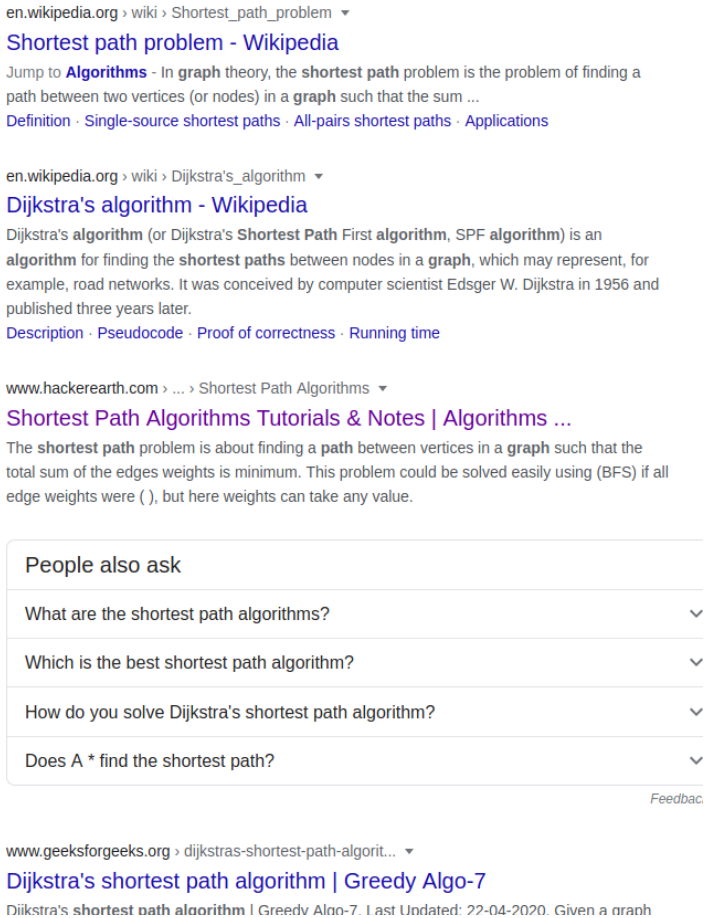
Jul 31, 2015 - Don't take our word for it – here's what the **audience** and critics are ... costu fit for a Tarsem Singh film, **Puccini** earworms & killer vocals.

operawire.com › no-hero-why-calaf-is-the-dramatic-pr...

No Hero: Why Calaf Is The Dramatic Problem In Puccini's ...

➥Close, but not quite the information I was looking for

Query: shortest path algorithms



en.wikipedia.org › wiki › Shortest_path_problem

Shortest path problem - Wikipedia

Jump to **Algorithms** - In **graph** theory, the **shortest path** problem is the problem of finding a path between two vertices (or nodes) in a **graph** such that the sum ...

Definition · Single-source shortest paths · All-pairs shortest paths · Applications

en.wikipedia.org › wiki › Dijkstra's_algorithm

Dijkstra's algorithm - Wikipedia

Dijkstra's **algorithm** (or Dijkstra's **Shortest Path** First **algorithm**, SPF **algorithm**) is an **algorithm** for finding the **shortest paths** between nodes in a **graph**, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

Description · Pseudocode · Proof of correctness · Running time

www.hackerearth.com › ... › Shortest Path Algorithms

Shortest Path Algorithms Tutorials & Notes | Algorithms ...

The **shortest path** problem is about finding a **path** between vertices in a **graph** such that the total sum of the edges weights is minimum. This problem could be solved easily using (BFS) if all edge weights were ( ), but here weights can take any value.

People also ask

What are the shortest path algorithms?

Which is the best shortest path algorithm?

How do you solve Dijkstra's shortest path algorithm?

Does A * find the shortest path?

Feedbac

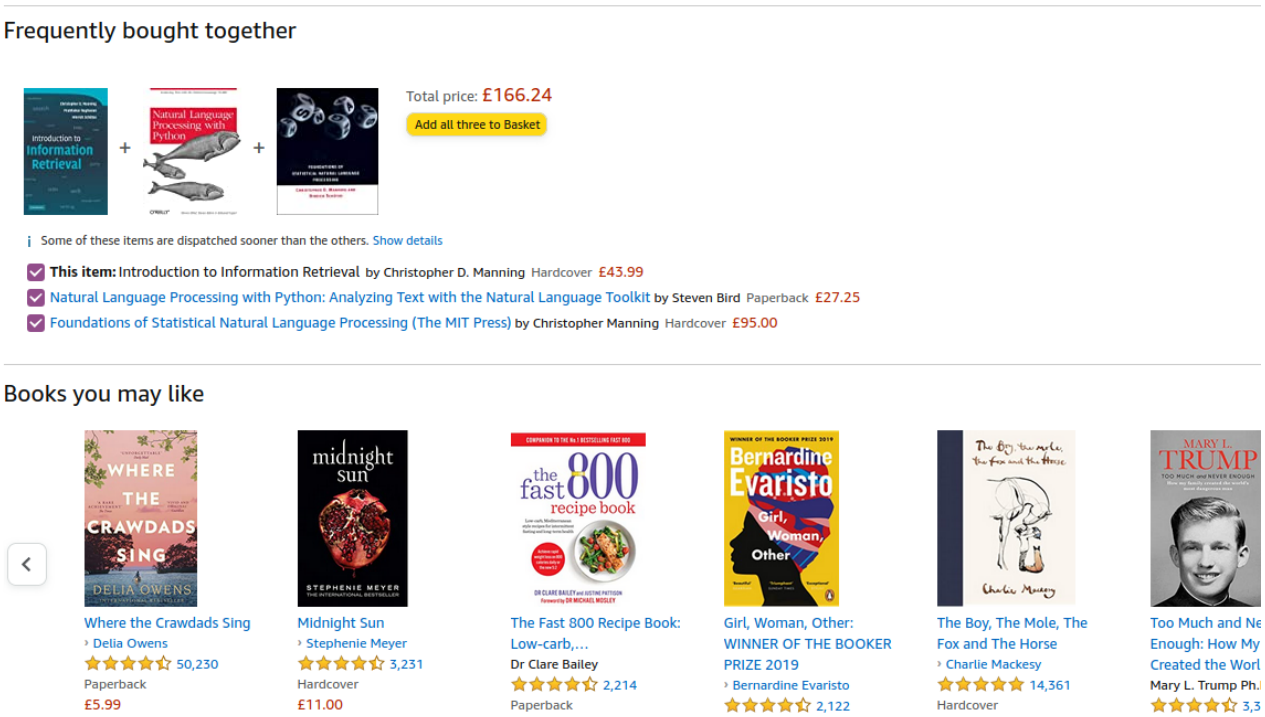
www.geeksforgeeks.org › dijkstras-shortest-path-algorit...

Dijkstra's shortest path algorithm | Greedy Algo-7

Dijkstra's **shortest path algorithm** | Greedy Algo-7. Last Updated: 22-04-2020. Given a graph

➥Relevant and competent resources

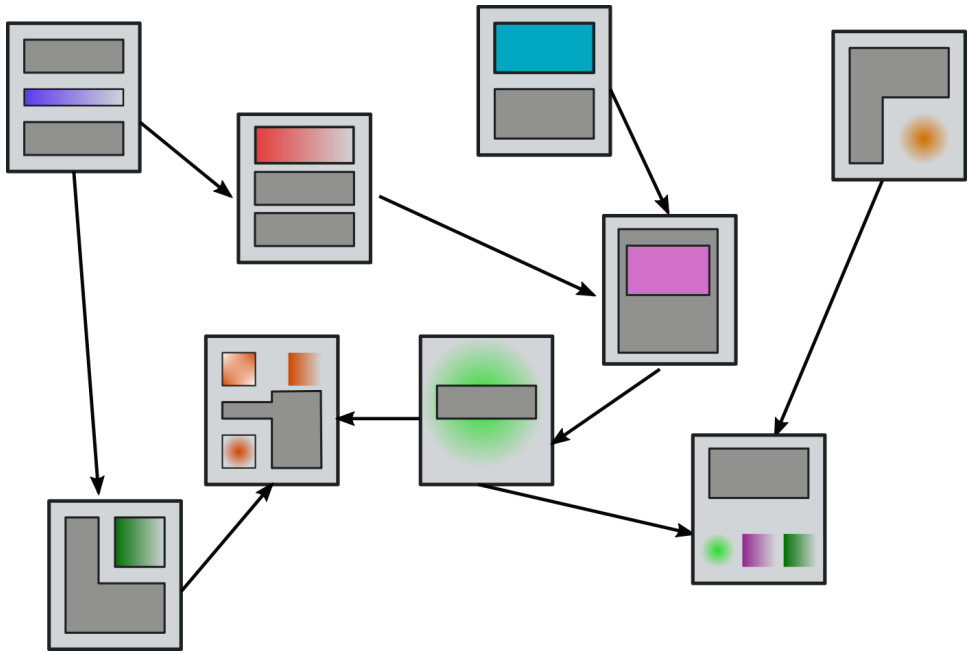
Looking on Amazon for the Manning IR book:



Two kinds of recommendations:

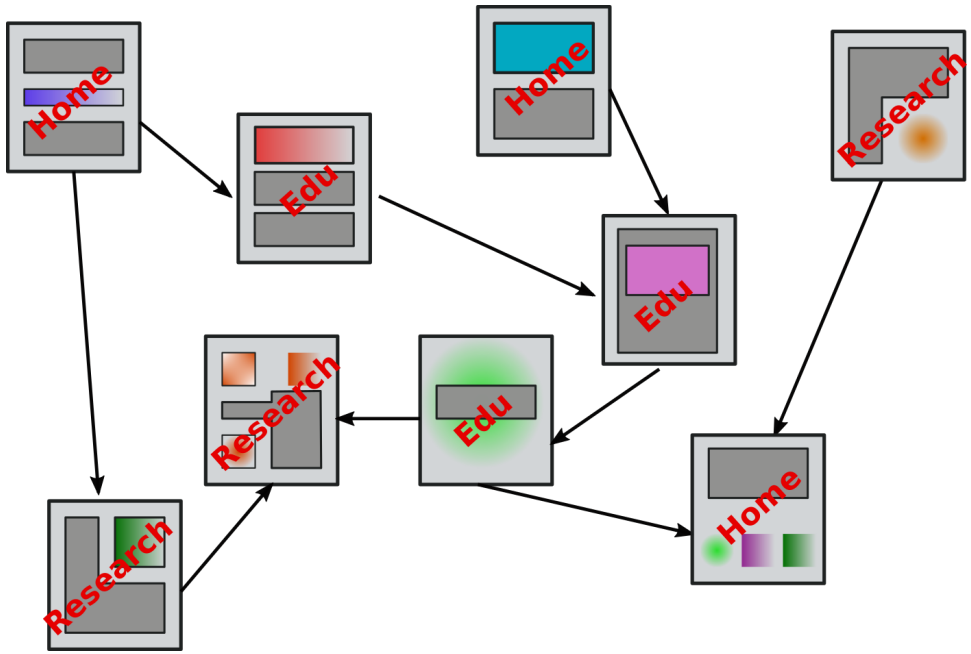
- Product-based: “similar” products (books)
- User-based: what I may like (?)

Suppose this is the `aau.dk` domain:



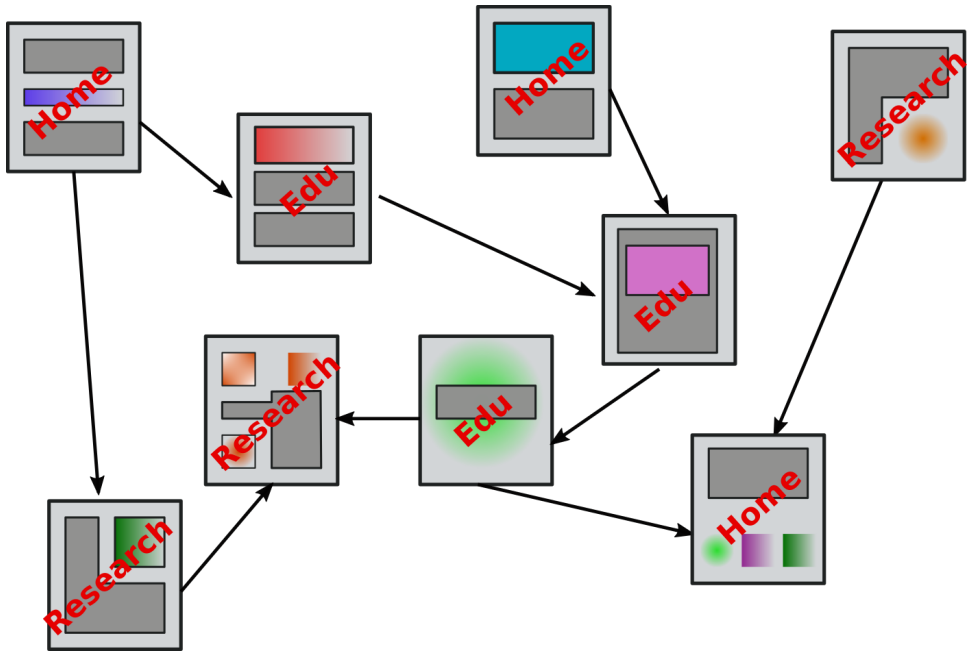
Which of the pages are *personal homepages*, which are about *educational programs*, and which are about *research projects*?

Suppose this is the `aau.dk` domain:



Which of the pages are *personal homepages*, which are about *educational programs*, and which are about *research projects*?

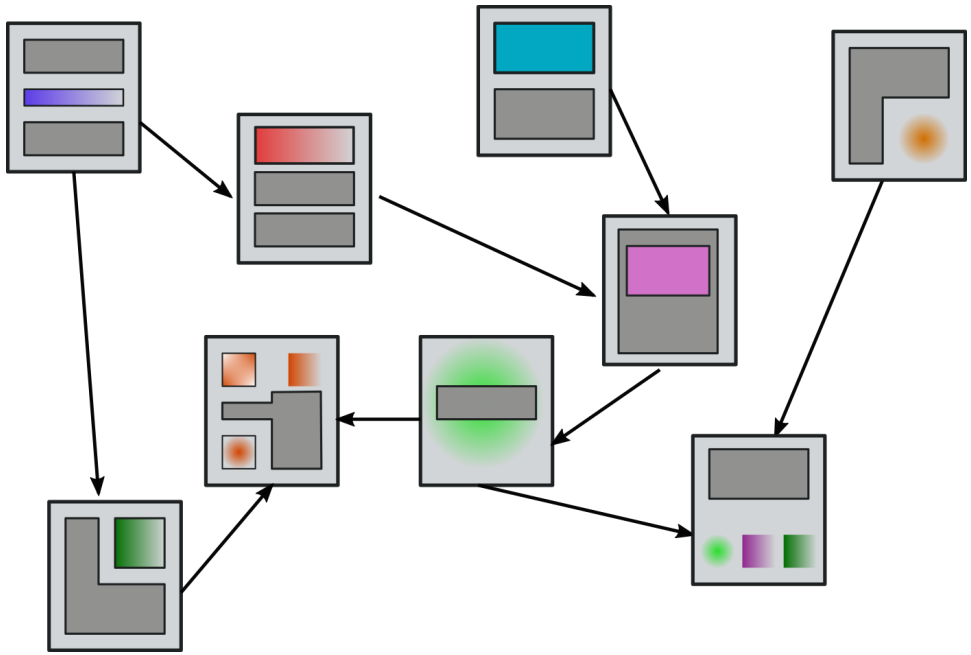
Suppose this is the `aau.dk` domain:



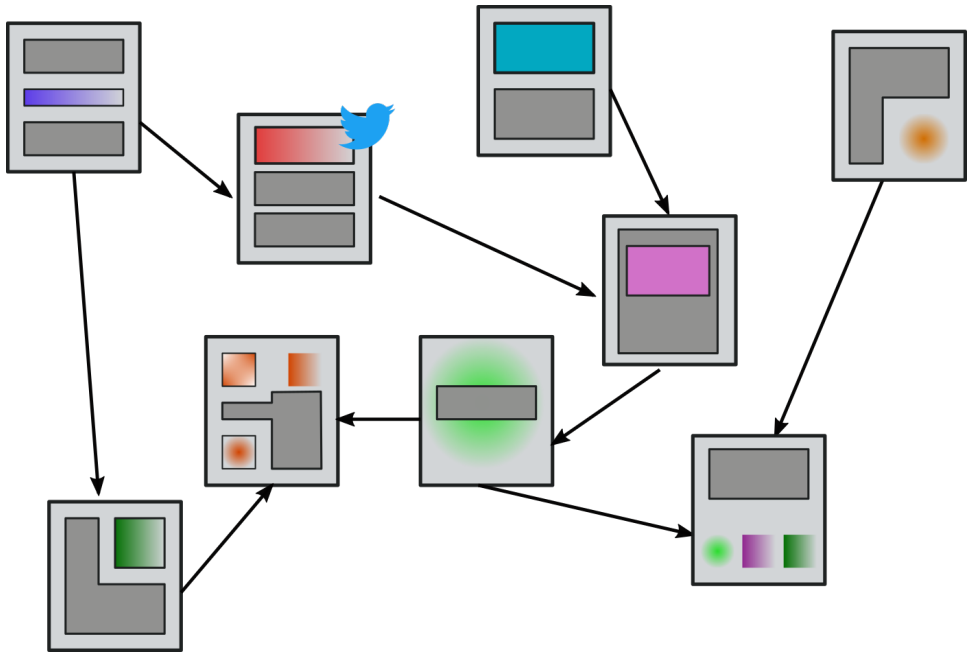
Which of the pages are *personal homepages*, which are about *educational programs*, and which are about *research projects*?

- Prediction models based on a node's properties, the properties of its neighbors, the neighbors' neighbors, . . .

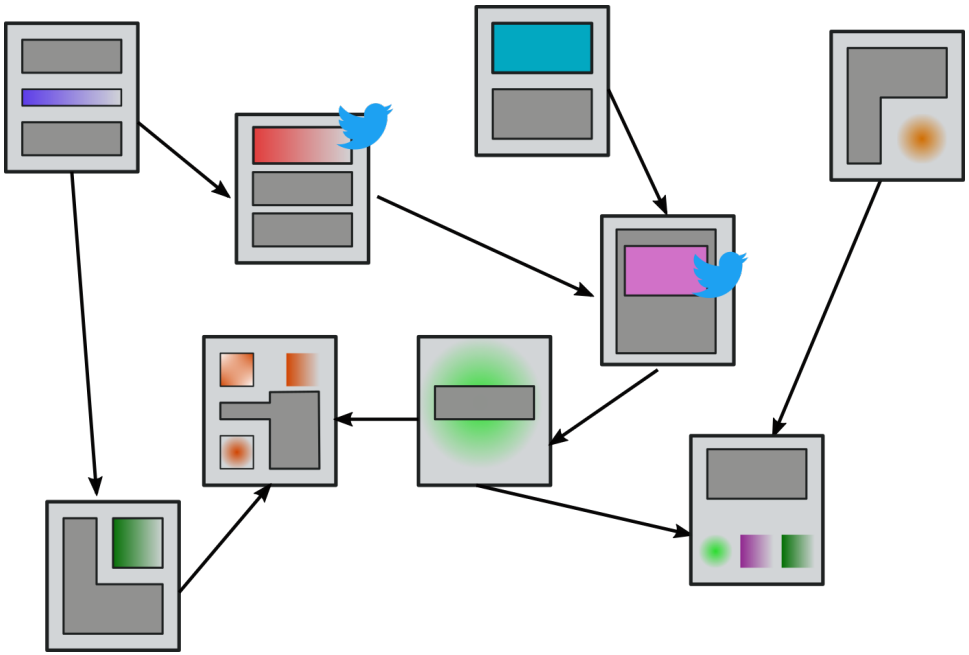
How does a piece of gossip, a virus, a “likes”, a re-tweet, ..., spread over a network?



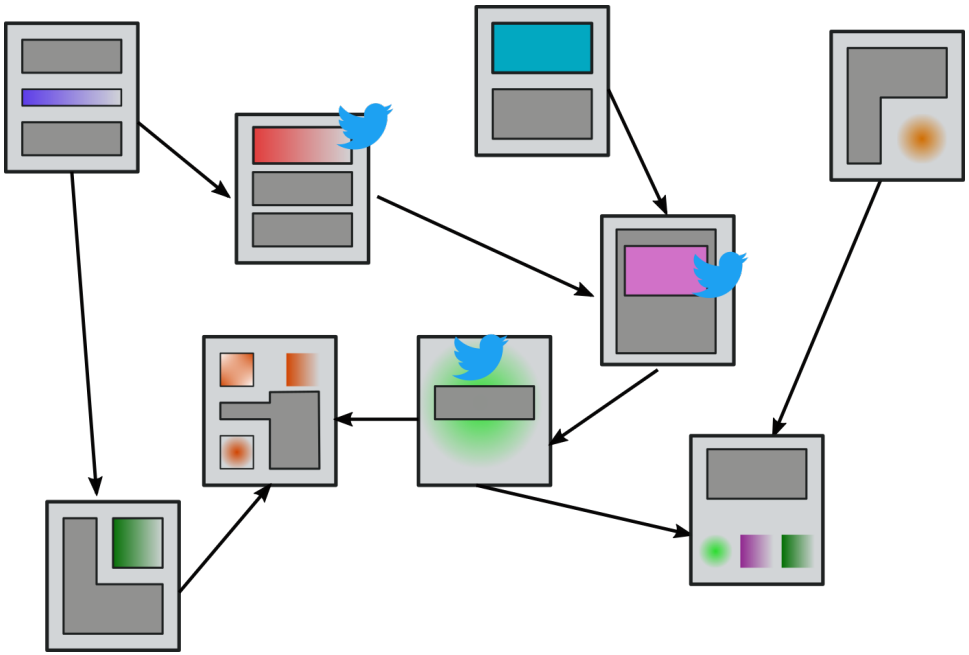
How does a piece of gossip, a virus, a “likes”, a re-tweet, ..., spread over a network?



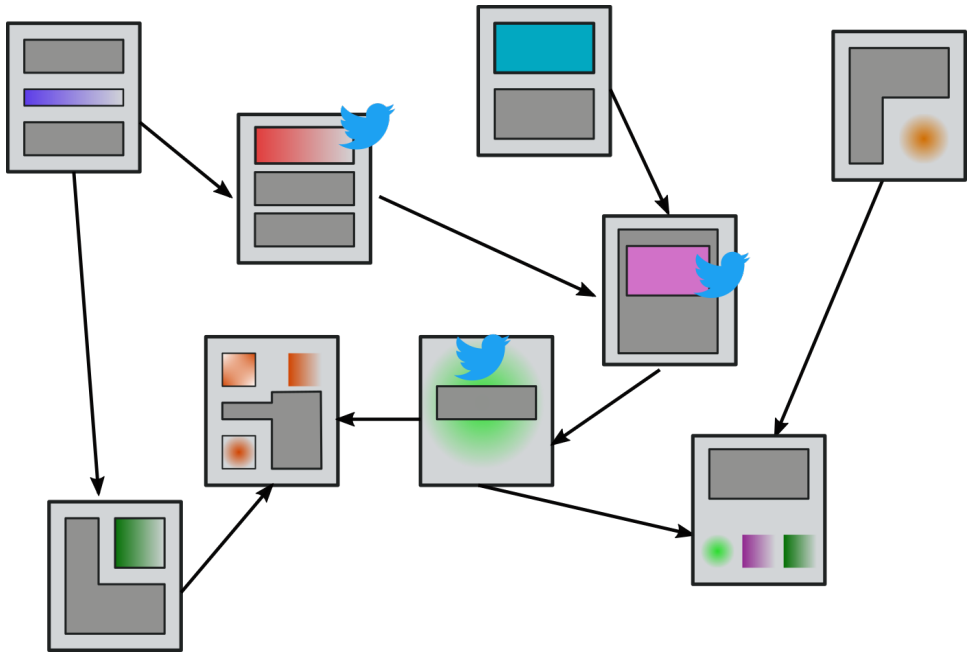
How does a piece of gossip, a virus, a “likes”, a re-tweet, ..., spread over a network?



How does a piece of gossip, a virus, a “likes”, a re-tweet, ..., spread over a network?



How does a piece of gossip, a virus, a “likes”, a re-tweet, ..., spread over a network?

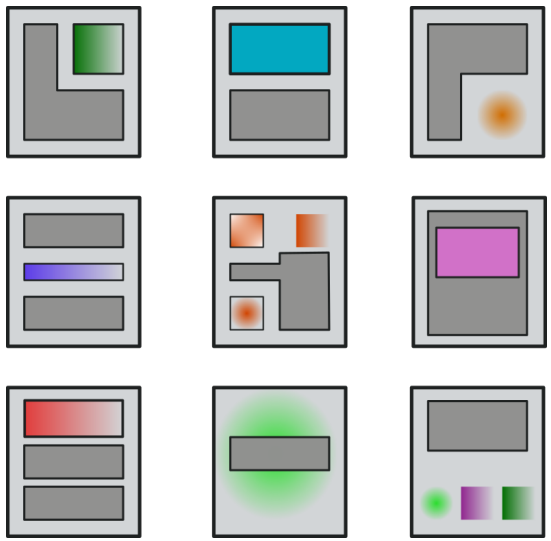


- Models for “information diffusion”
- Use for prediction of information spread, identification of effective “seed nodes”, . . .

We can distinguish 3 levels (perspectives) of modeling and analytics:

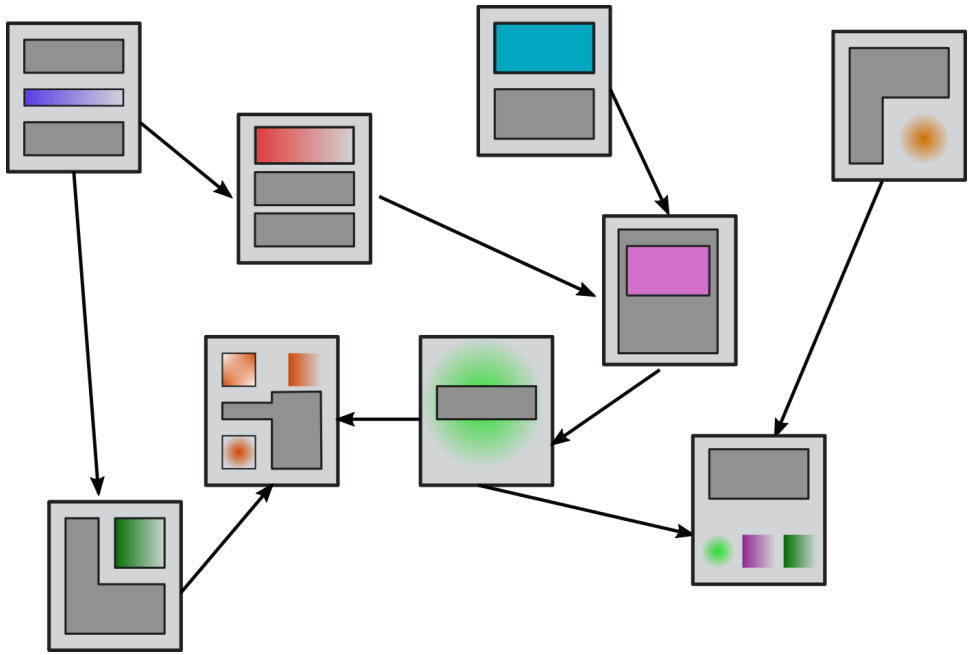
Level 1: Web Content

The web as a collection of documents:



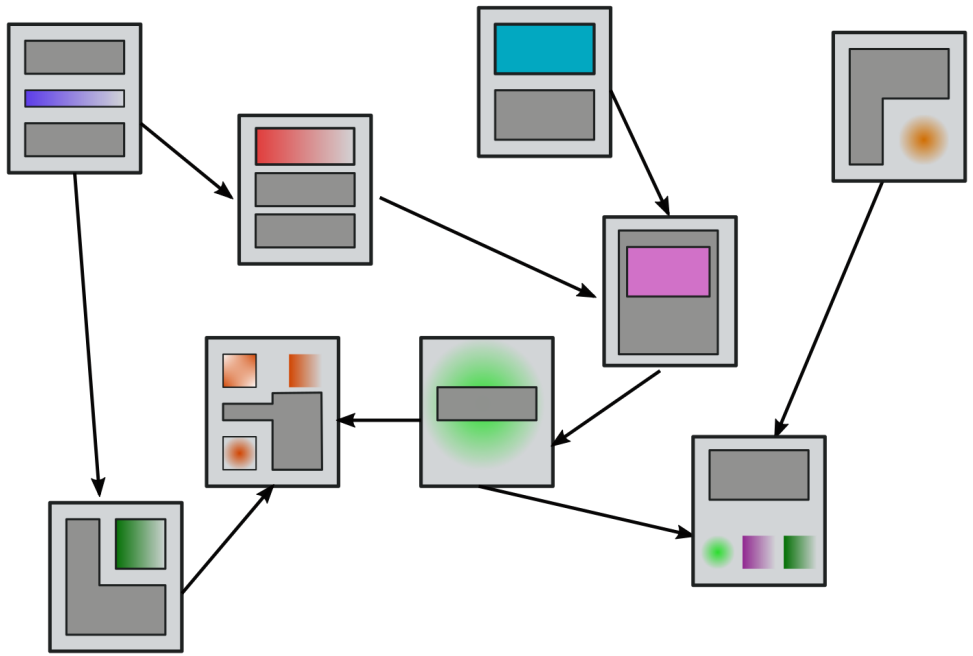
Level 2: Web Structure

The web as a network of documents:



Level 3: Web Dynamics

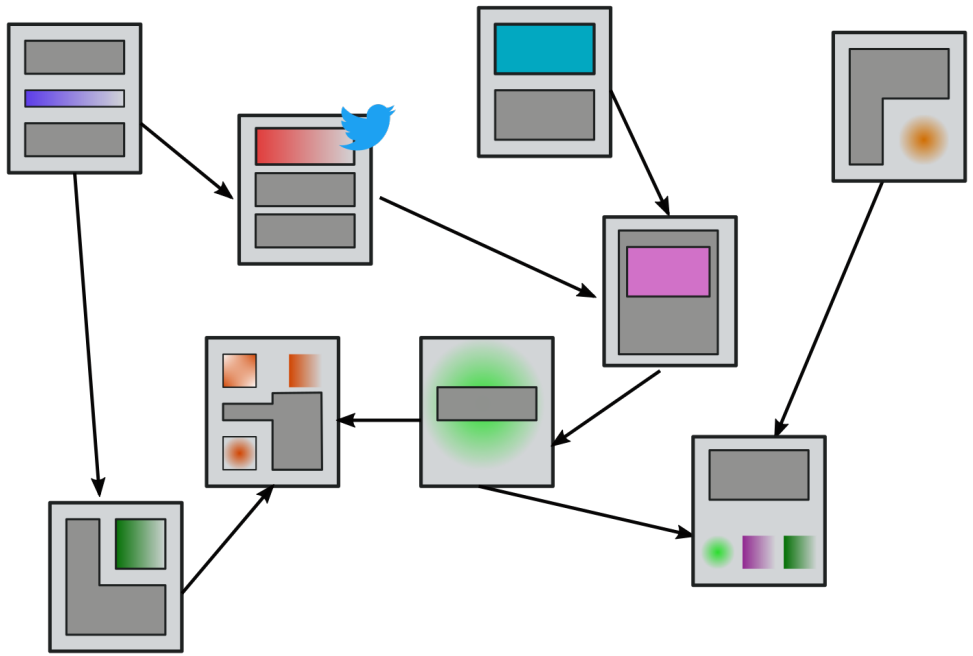
The web as a dynamic network:



- Network evolution
- Dynamic processes on the web

Level 3: Web Dynamics

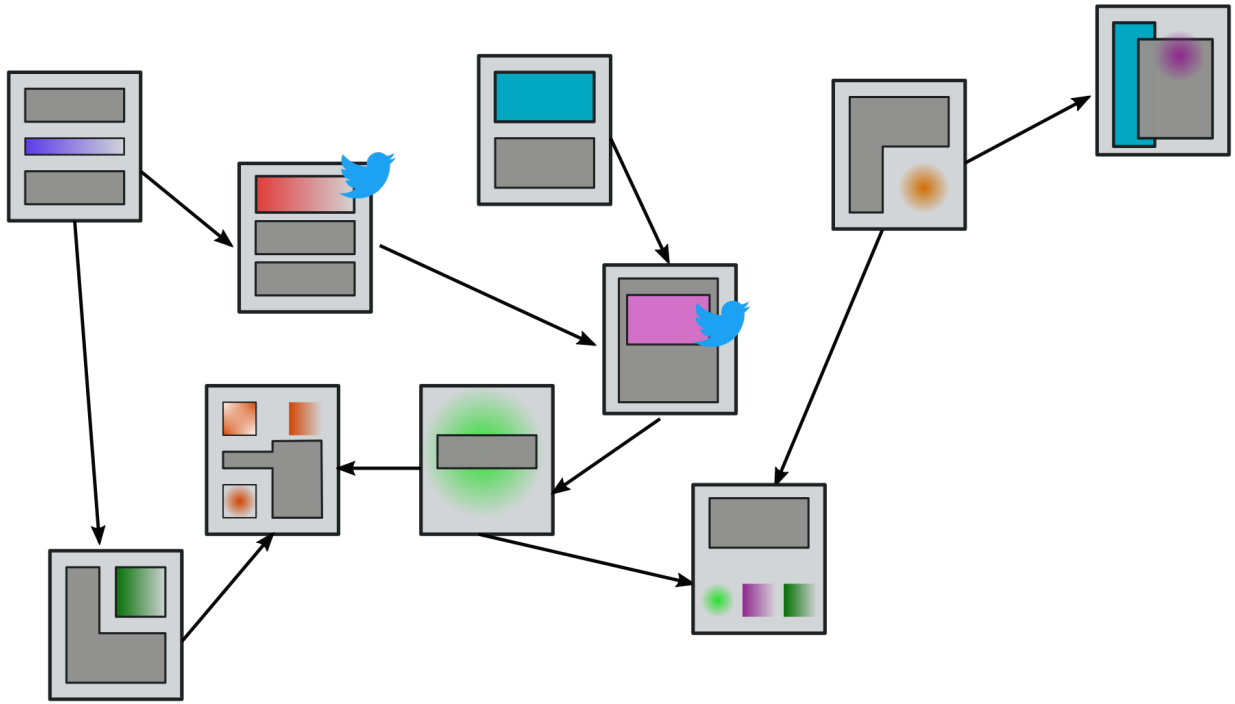
The web as a dynamic network:



- Network evolution
- Dynamic processes on the web

Level 3: Web Dynamics

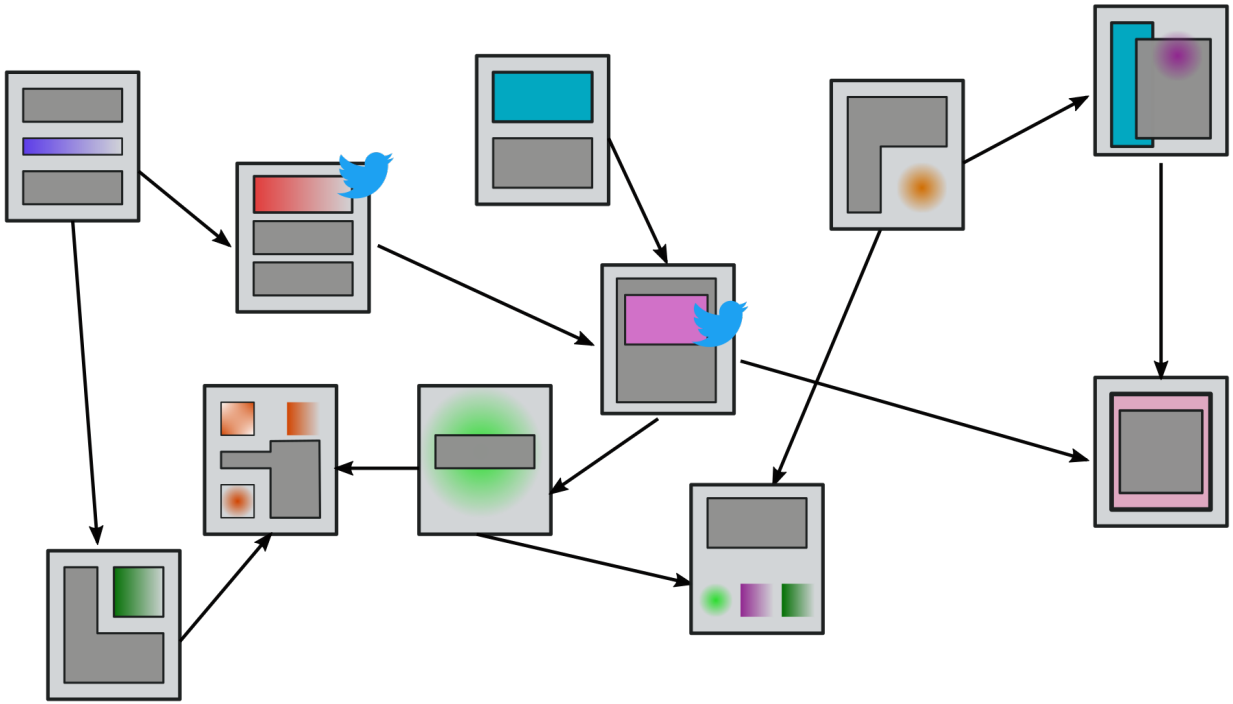
The web as a dynamic network:



- Network evolution
- Dynamic processes on the web

Level 3: Web Dynamics

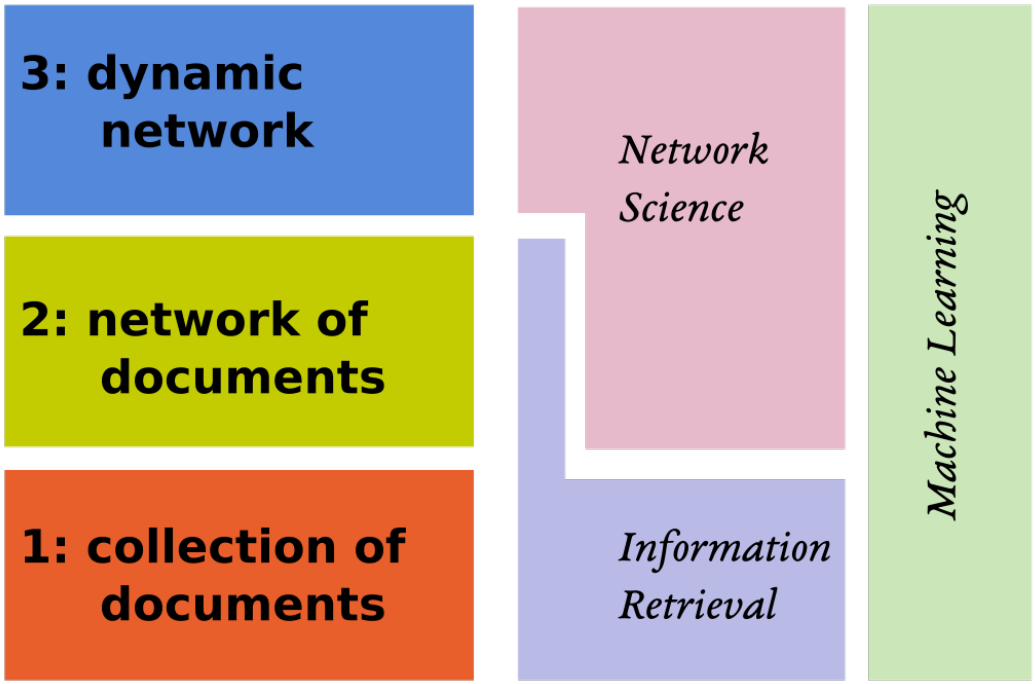
The web as a dynamic network:



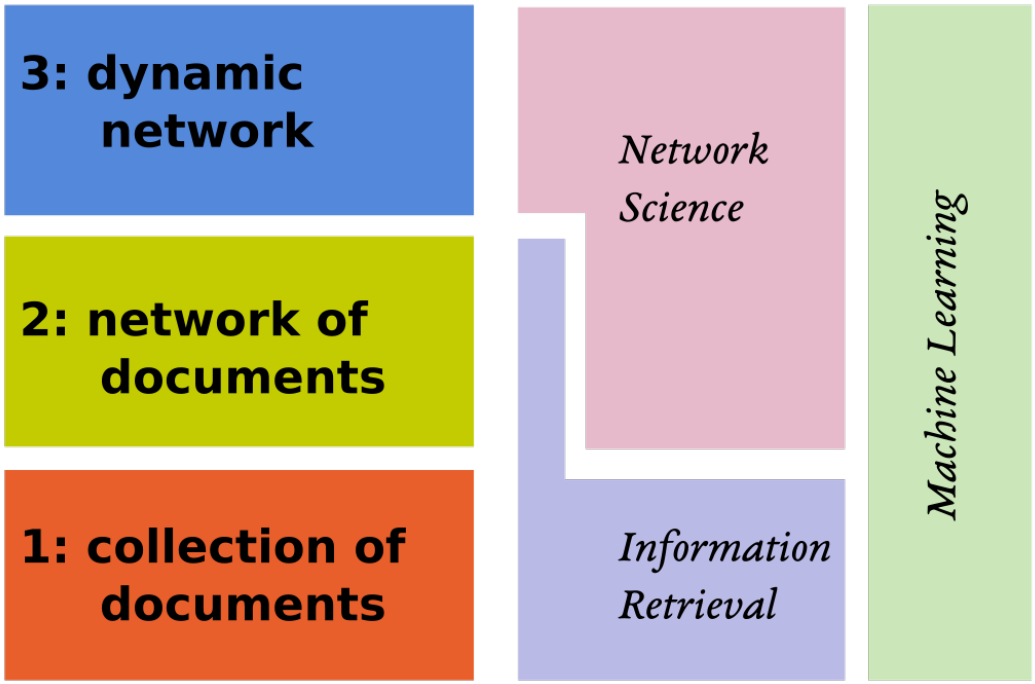
- Network evolution
- Dynamic processes on the web

Relationship to scientific disciplines:

Relationship to scientific disciplines:



Relationship to scientific disciplines:

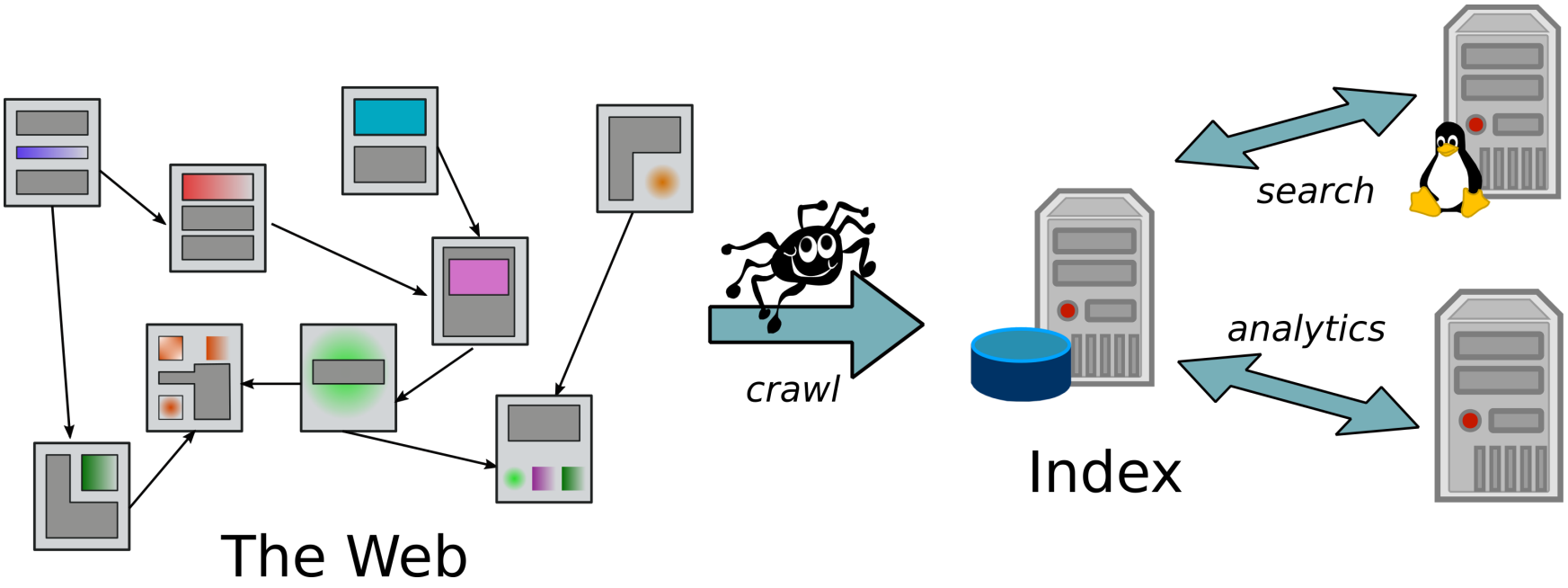


Network science: mostly based on a pure “graph view” (nodes + edges) of networks.

➥Heterogeneous networks of “multi-media nodes” still a source of many research challenges

Web Crawlers

Before we can do anything, web data needs to be retrieved and organized:



Credits: clipart from pixabay.com

Skeleton structure of a crawler:

```
CRAWL(URL set: seeds)
1 frontier=seeds
2 while frontier≠ ∅ do
3 |   url= get_url(frontier)              // select next URL from frontier
4 |   doc= fetch(url)          // pages returned as html source text documents
5 |   index(doc)                           // send doc to indexer
6 |   frontier.add(extract_urls(doc))
7 end
```

Key design issue: the *frontier* of URLs to be processed, and selection strategy implementing *get_url*.

Two simplistic solutions:

- *frontier* as stack. Leads to depth-first search. Problem: can get quickly stuck in “dead end” remote corners of the web
- *frontier* as queue. Leads to breadth-first search. Problem: slow progress, lacking politeness (see below)

Both are too simple, because:

- A pure sequential, single thread architecture will get stuck once a host does not respond (quickly) to a *fetch*(*url*) request
- Crawler must implement **robustness**: not get stuck in *spider traps*, i.e., large, dead-end (uninteresting) web components
- Crawler must implement **politeness**: not overload a single web server with requests

Crawlers can have different purposes:

- *Periodic*: maintain an up-to-date general picture of the web.
  - The same pages (URLs) should be re-visited periodically
- *Focused*: map a part of the web pertaining to a particular topic

Implemented by:

- *index*(*doc*): maybe not every fetched document needs to be added to the index
- *frontier.add*(*extract_urls*(*doc*)): extracted URLs may be added to the frontier with different **priorities**:
  - URLs that have already been recently visited have a lower priority (periodic crawling)
  - URLs that are less likely to refer to relevant pages have a lower priority (focused crawling)
  - ...

- Minimum time delay between two requests to one host
- Obey `robots.txt` file

robots.txt

Text file at top level of domain: `http://domain.com/robots.txt`. Provides instructions to crawlers.

Don't allow any crawlers to go to `/private/` directory:

```
User-agent:  *
Disallow:  /private/
```

Allow all crawlers all access, except googlebot is not allowed in `/tmp/`:

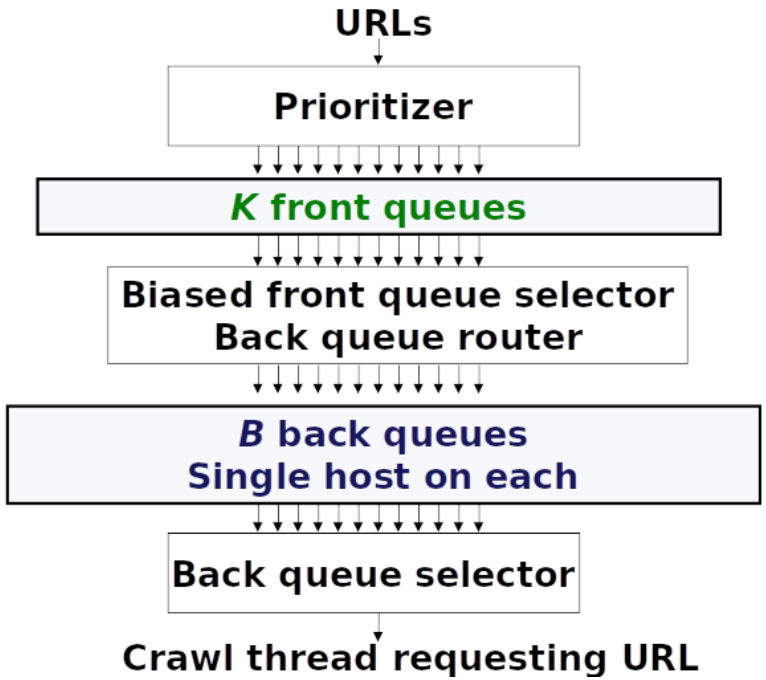
```
User-agent:  *
Disallow:
User-agent:  googlebot
Disallow:  /tmp/
```

Non-standard extension of `robots.txt`: request delay between successive visits:

```
User-agent:  bingbot
Crawl-delay:  5
```

The interpretation of the delay values can be crawler specific.

Heydon, A., & Najork, M. (1999). Mercator: A scalable, extensible web crawler. World Wide Web, 2(4), 219-229.

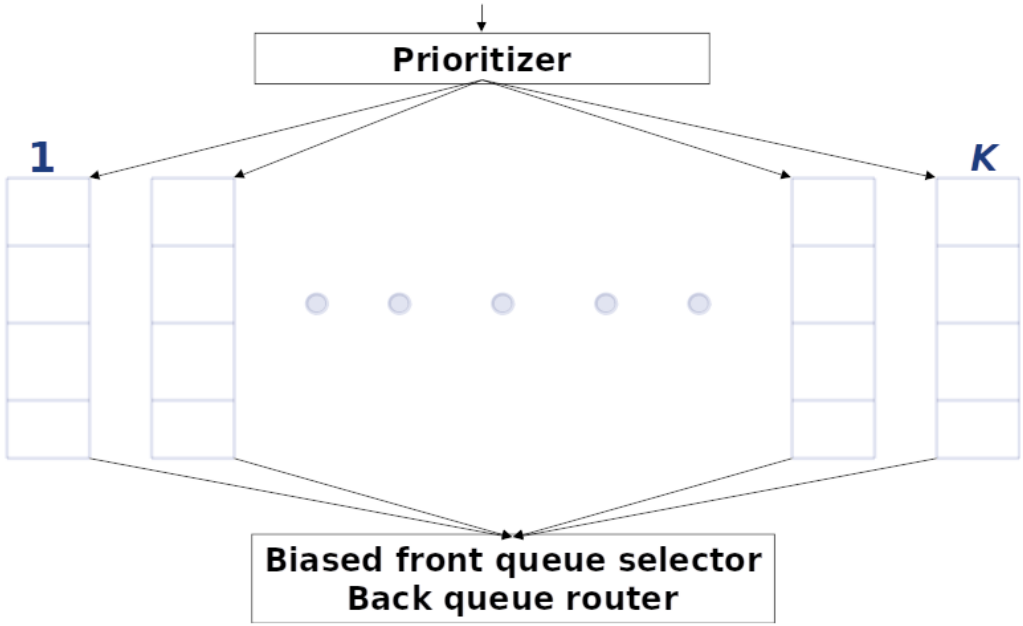


Front queues: for prioritization

Back queues: for politeness

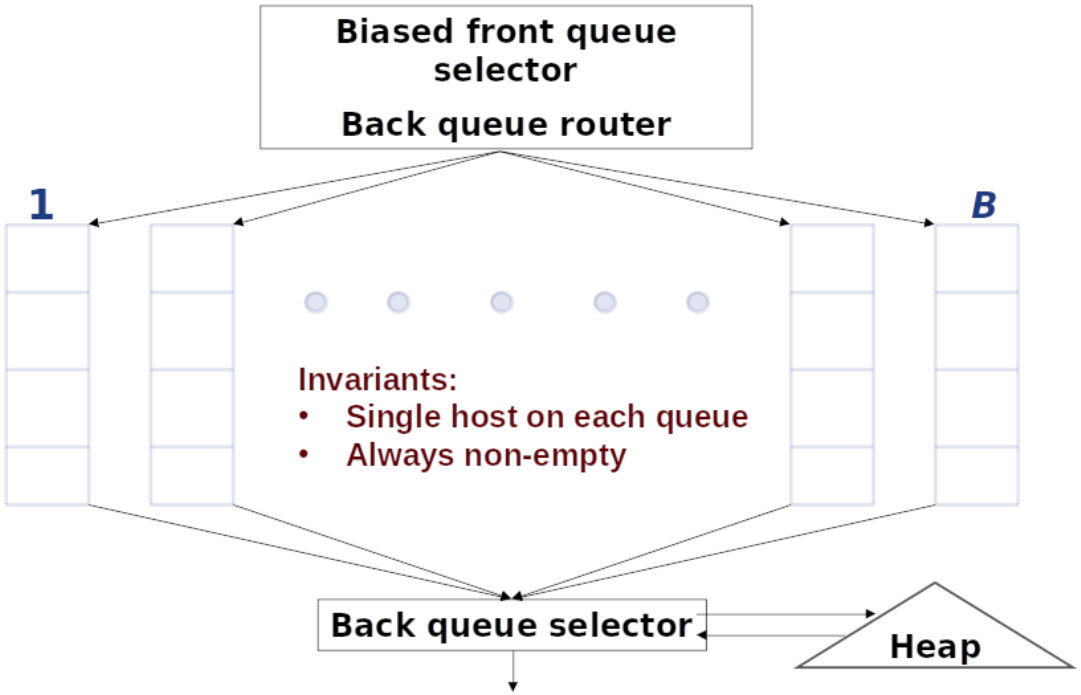
This and following slides: images derived from lecture slides of Chris Manning and Pandu Nayak

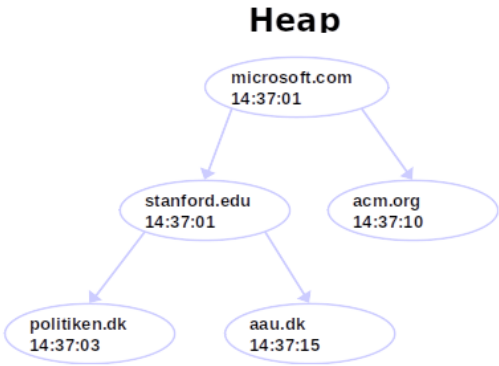
Fixed number of K FIFO queues:



- Incoming URLs are assigned a priority value between 1 and K, and enqueued in the corresponding front queue.
- URLs are extracted by
  - Selecting (e.g. randomized) one of the front queues; higher priority queues are more likely to be selected
  - Dequeueing the head element from the selected queue

Fixed number of *B* FIFO queues:





Current host to back queue mapping

Host name	Back queue
cs.aau.dk	3
microsoft.com	1
...	...
acm.org	*B*

- Each back queue contains URLs from only one host
- Each queue/host has an entry in a priority queue (heap) that determines from which back queue the next URL will be extracted
- Priority value: time stamp at which next request to host can be made at the earliest (following politeness policy)

Getting the next URL:

- Determine highest priority host
- Dequeue head element from corresponding queue
- Update priority value of host

If queue of selected host becomes empty, re-fill back-queues from front queues as follows:

- Get next URL *url* from front queues
- If *url*'s host already has a back queue: enqueue there
- Otherwise:
  - enqueue *url* in the empty queue
  - update heap and host dictionary
- Repeat until queue non-empty

In practice: use multiple crawlers

- Each crawler has its own URL frontier
- URLs are distributed over crawlers according to host: each crawler is responsible for a certain set of hosts (e.g. defined by a hash function, or geographically)
- The

  frontier.add(extract_urls(doc))

  operation must add URLs to the frontier of the relevant crawler

Duplicate Identification

Many web-pages are duplicates or near-duplicates of other pages:

- Mirrors
- Identical product descriptions, user manuals, etc. contained on diverse web sites
- ...

Estimate: as many as 40% of pages have duplicates [Manning et al., 2009]

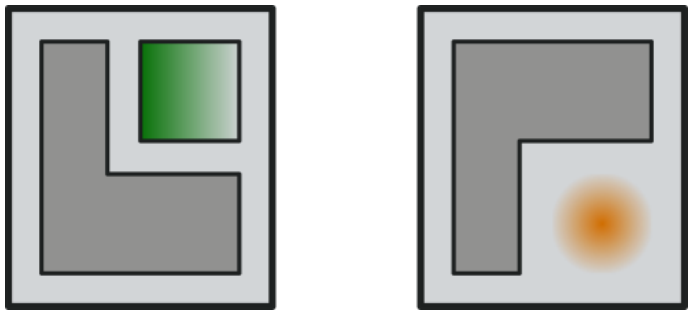
➥May not want to include all duplicates in index

➥The

index(doc)

operation may include a prior test whether *doc* is a (near-)duplicate of an already indexed page

Are these near duplicates?



➥Will only consider the texts of the pages!

Fingerprints

If we wanted to detect *identical* texts, things would be relatively easy: construct a *hash code*

$$\textit{text} \rightarrow \text{64bit integers}$$

such that non-identical texts are unlikely to be mapped to the same integer ($1.8 \cdot 10^{19}$ codes vs. $\sim 10^{11}$ web documents).

k-Shingles, or k-Grams

We view text as a **set** of consecutive sequences of k words:

we view text as a sequence of words
we view text as
view text as a
text as a sequence
as a sequence of
a sequence of words

4-shingle representation:

{ as a sequence of, a sequence of words, text as a sequence, view text as a, we view text as }

- Order of occurrence of shingles is not included in representation
- Number of occurrences of a shingle is not included in representation
- Some pre-processing of raw html text before shingling (e.g.: ignore case, remove html tags)

Assuming a fixed vocabulary of size N, there are N^4 different 4-shingles, and we can identify them with the integers $0, \ldots, N^4 - 1$.

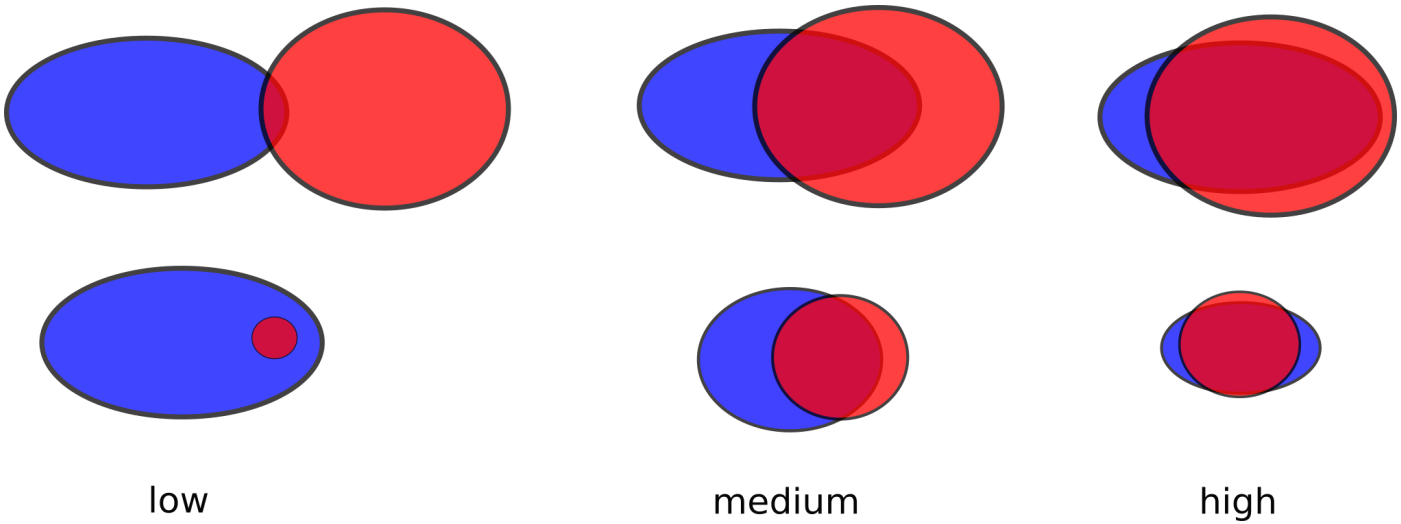
[Broder, Andrei Z., et al. "Syntactic clustering of the web." Computer networks and ISDN systems 29.8-13 (1997): 1157-1166.]

Jaccard Coefficient

For *any* two finite sets A, B, define

$$J(A, B) := \frac{|A \cap B|}{|A \cup B|}$$

$J(A, B)$ measures the overlap relative to the total size of the sets:



We measure the similarity of text documents d_1, d_2 by the Jaccard coefficient of their shingle sets $S(d_1), S(d_2)$.

Assume documents are represented by their (sorted) sets of shingle indices in $0, \ldots, N^4 - 1$:

$$\begin{aligned} S(d_1): &\quad \{3, 36, 1834, 1947, \ldots, 4982840\} \\ S(d_2): &\quad \{18, 54, 1834, 21895, \ldots, 5004298\} \end{aligned}$$

Naive way to compute $|S(d_1) \cap S(d_2)|$ and $|S(d_1) \cap S(d_2)|$: step through both lists and count number of common entries and total number of distinct entries.

Complexity: linear in the maximum length of the two documents.

➥not good enough!

Estimating $J(S(d_1), S(d_2))$:

- Let π be a random permutation of the integers $0, \ldots, N^4 - 1$.
- For $j = 1, 2$: let $x_j^\pi := min\{\pi(x) : x \in S(d_j)\}$

➥ Then:

$$J(S(d_1), S(d_2)) = P(x_1^\pi = x_2^\pi),$$

where P is the probability over the selection of a random permutation π.

Estimating $J(S(d_1), S(d_2))$:

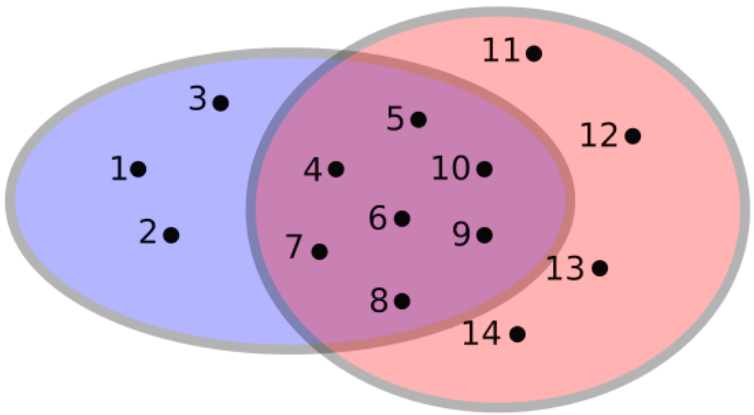
- Let π be a random permutation of the integers $0, \ldots, N^4 - 1$.
- For $j = 1, 2$: let $x_j^\pi := min\{\pi(x) : x \in S(d_j)\}$

➥ Then:

$$J(S(d_1), S(d_2)) = P(x_1^\pi = x_2^\pi),$$

where P is the probability over the selection of a random permutation π.

Illustration (for simplicity: $S(d_1) \cup S(d_2) = \{1, 2, \ldots, 14\}$):



Estimating $J(S(d_1), S(d_2))$:

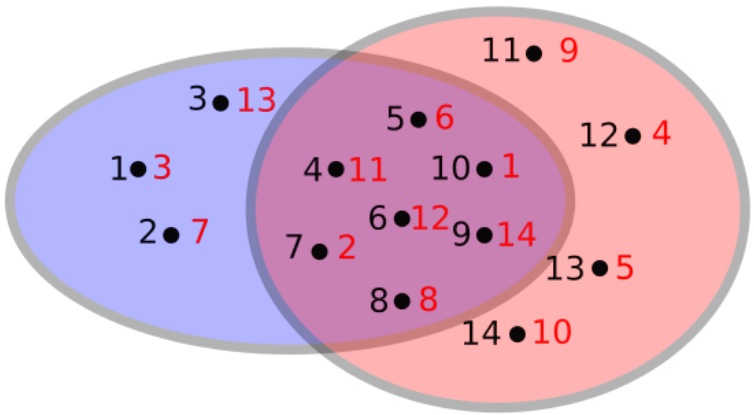
- Let π be a random permutation of the integers $0, \ldots, N^4 - 1$.
- For $j = 1, 2$: let $x_j^\pi := min\{\pi(x) : x \in S(d_j)\}$

➥ Then:

$$J(S(d_1), S(d_2)) = P(x_1^\pi = x_2^\pi),$$

where P is the probability over the selection of a random permutation π.

Illustration (for simplicity: $S(d_1) \cup S(d_2) = \{1, 2, \ldots, 14\}$):



$\min \pi(S(d_1) \cup S(d_2)) \in S(d_1) \cap S(d_2)$

Estimating $J(S(d_1), S(d_2))$:

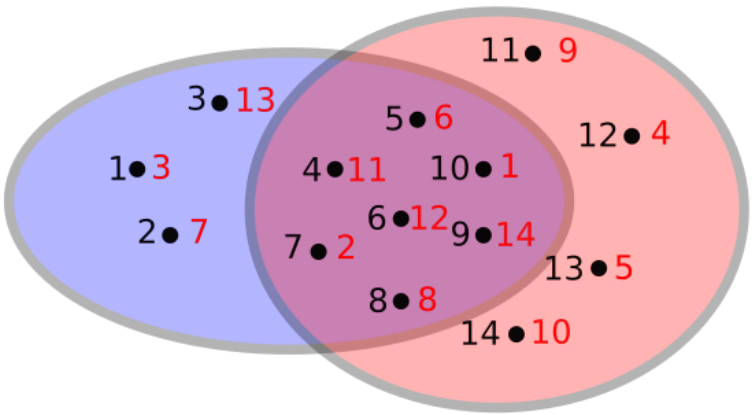
- Let π be a random permutation of the integers $0, \ldots, N^4 - 1$.
- For $j = 1, 2$: let $x_j^\pi := min\{\pi(x) : x \in S(d_j)\}$

➥ Then:

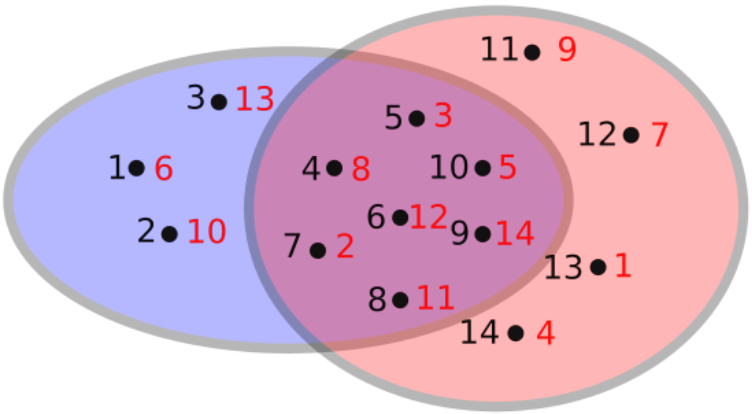
$$J(S(d_1), S(d_2)) = P(x_1^\pi = x_2^\pi),$$

where P is the probability over the selection of a random permutation π.

Illustration (for simplicity: $S(d_1) \cup S(d_2) = \{1, 2, \ldots, 14\}$):



$\min \pi(S(d_1) \cup S(d_2)) \in S(d_1) \cap S(d_2)$



$\min \pi(S(d_1) \cup S(d_2)) \notin S(d_1) \cap S(d_2)$

Estimating $J(S(d_1), S(d_2))$:

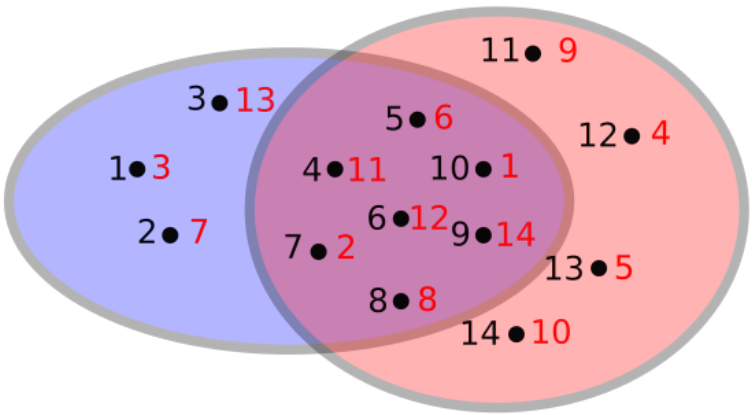
- Let π be a random permutation of the integers $0, \ldots, N^4 - 1$.
- For $j = 1, 2$: let $x_j^\pi := min\{\pi(x) : x \in S(d_j)\}$

➥ Then:

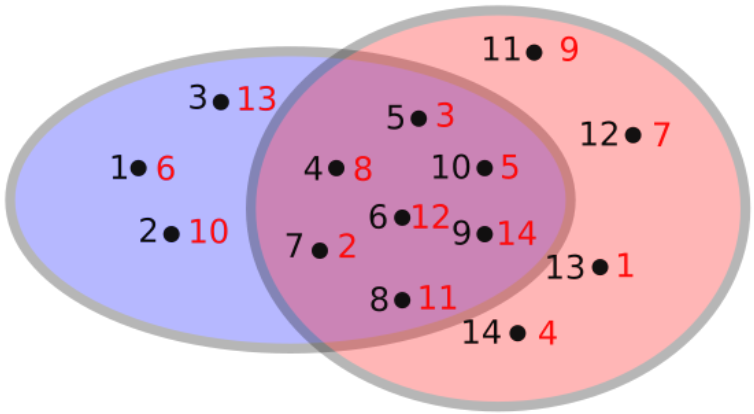
$$J(S(d_1), S(d_2)) = P(x_1^\pi = x_2^\pi),$$

where P is the probability over the selection of a random permutation π.

Illustration (for simplicity: $S(d_1) \cup S(d_2) = \{1, 2, \ldots, 14\}$):



$\min \pi(S(d_1) \cup S(d_2)) \in S(d_1) \cap S(d_2)$



$\min \pi(S(d_1) \cup S(d_2)) \notin S(d_1) \cap S(d_2)$

- $P(x_1^\pi = x_2^\pi)$ = probability that the minimum value $min\,\pi(S(d_1) \cup S(d_2))$ falls inside $S(d_1) \cap S(d_2) = J(S(d_1), S(d_2))$

One random permutation does not tell us much, so we take many, e.g.: $\pi_1, \pi_2, \ldots, \pi_{200}$.

Then characterize every document d_h by its feature vector

$$\psi(d_h) := (x_h^{\pi_1}, \ldots, x_h^{\pi_{200}}),$$

called the **sketch** of d_h:

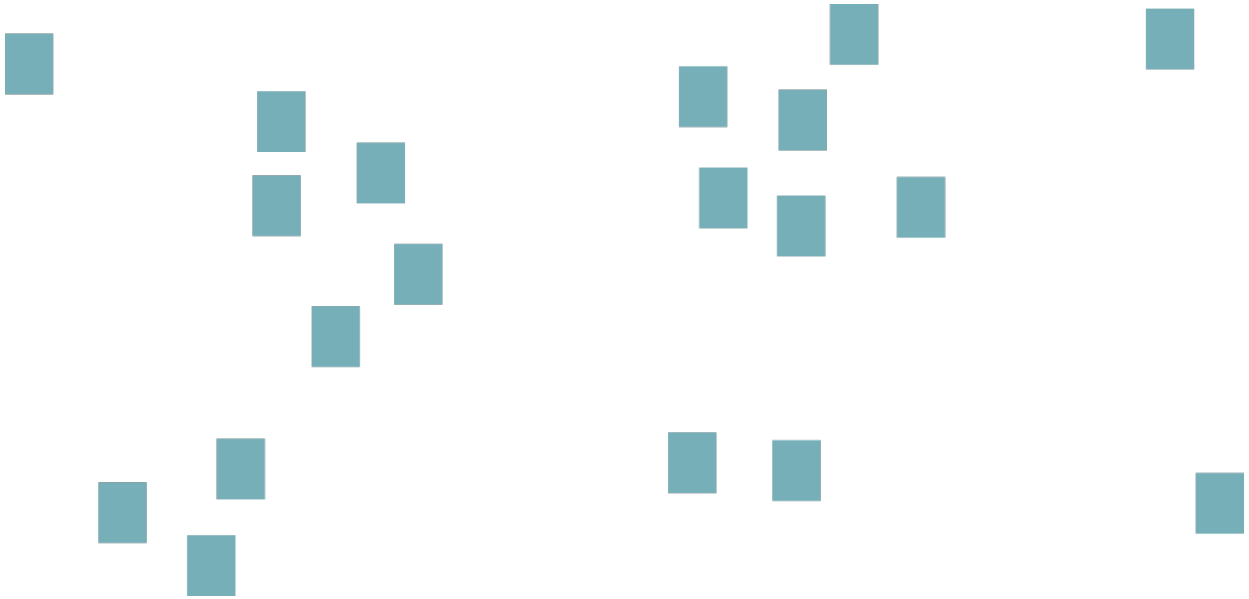
d	$\psi(d) =$			
	x^{π_1}	x^{π_2}	$\ldots$	$x^{\pi_{200}}$
d_1	1965	20743	$\ldots$	4975
d_2	32987	20743	$\ldots$	11764
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
d_h	1965	984	$\ldots$	4975
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Now we can *approximately estimate* the Jaccard coefficient:

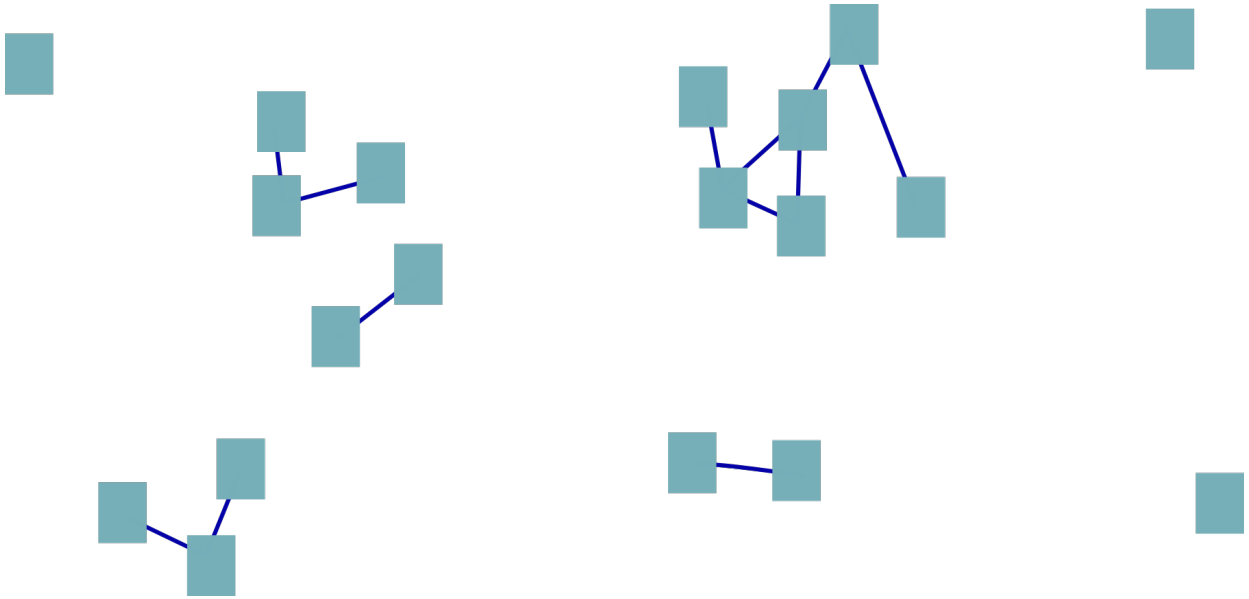
$$J(S(d_1), S(d_2)) \approx \frac{\#\text{equal components in } \psi(d_1) \text{ and } \psi(d_2)}{200}$$

➥Complexity no longer depends on the lengths of the document.

Given a collection of documents:



Given a collection of documents:

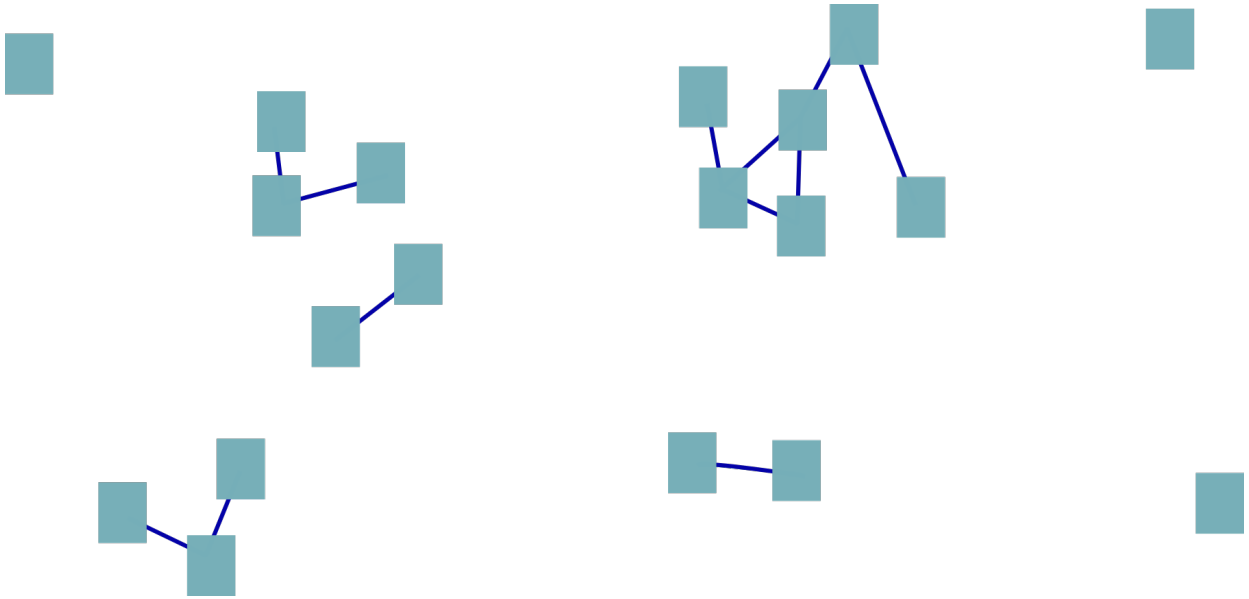


Group documents into **clusters**, so that near-duplicates are in one cluster

Supports:

In order to show you the most relevant results, we have omitted some entries very similar to the 110 already displayed.
If you like, you can repeat the search with the omitted results included.

Given a collection of documents:



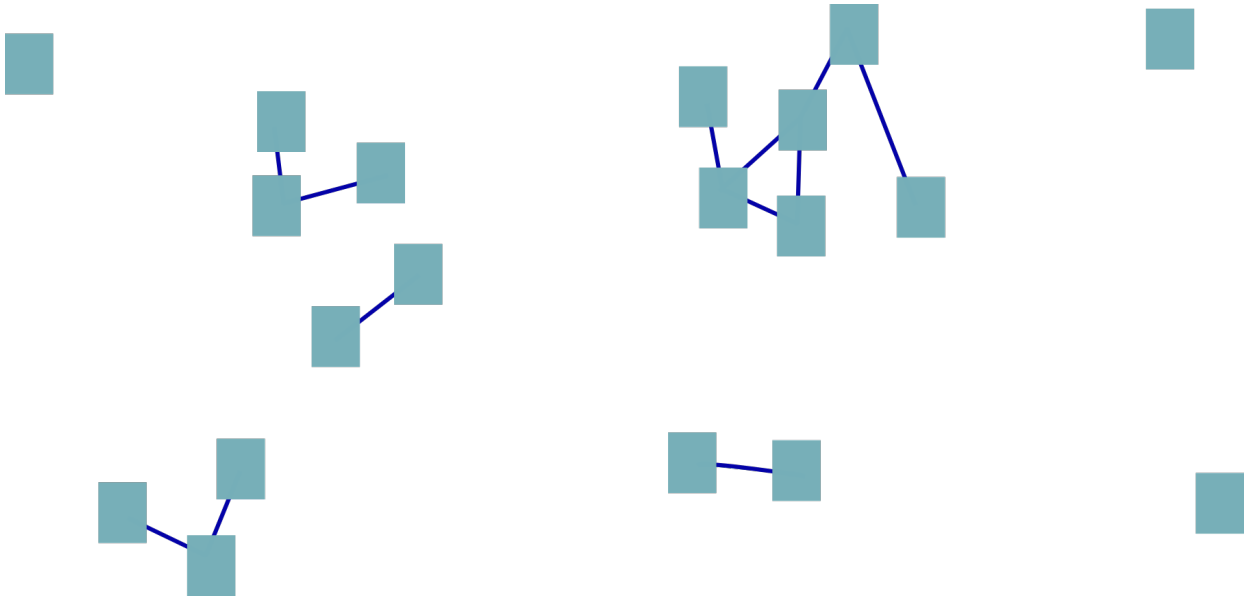
Group documents into **clusters**, so that near-duplicates are in one cluster

Supports:

In order to show you the most relevant results, we have omitted some entries very similar to the 110 already displayed.
If you like, you can repeat the search with the omitted results included.

➥Caution: this does not mean that all documents in a cluster are near-duplicates!

Given a collection of documents:



Group documents into **clusters**, so that near-duplicates are in one cluster

Supports:

In order to show you the most relevant results, we have omitted some entries very similar to the 110 already displayed.
If you like, you can repeat the search with the omitted results included.

➥Caution: this does not mean that all documents in a cluster are near-duplicates!

➥Will give useful results only if there are no long near-duplicate paths leading from one document to a totally different one

Naive **agglomerative single-link clustering** using a **union-find** data structure:

AggSLC(Document set: $\{d_1, \ldots, d_N\}$, Threshold t)

```
1 Initialize union-find DS in which every d_i is a singleton set
2 for all pairs d_i, d_j do
3 |   if J(S(d_i), S(d_j)) > t then
4 |   |   join the sets containing d_i and d_j
5 |   end
6 end
```

- line 3: can be approximated using sketches
- line 4: two *find* and (at most) one *union* operation

Naive **agglomerative single-link clustering** using a **union-find** data structure:

AggSLC(Document set: $\{d_1, \ldots, d_N\}$, Threshold t)

```
1 Initialize union-find DS in which every d_i is a singleton set
2 for all pairs d_i, d_j do
3 |   if J(S(d_i), S(d_j)) > t then
4 |   |   join the sets containing d_i and d_j
5 |   end
6 end
```

- line 3: can be approximated using sketches
- line 4: two *find* and (at most) one *union* operation

➥Complexity is $\Theta(N^2)$, which already is infeasible!

Naive **agglomerative single-link clustering** using a **union-find** data structure:

AggSLC(Document set: $\{d_1, \ldots, d_N\}$, Threshold t)

```
1 Initialize union-find DS in which every d_i is a singleton set
2 for all pairs d_i, d_j do
3 |   if J(S(d_i), S(d_j)) > t then
4 |   |   join the sets containing d_i and d_j
5 |   end
6 end
```

- line 3: can be approximated using sketches
- line 4: two *find* and (at most) one *union* operation

➥Complexity is $\Theta(N^2)$, which already is infeasible!

➥Basic strategy: filter out pairs d_i, d_j for which $J(S(d_i), S(d_j)) > t$ surely will not hold.

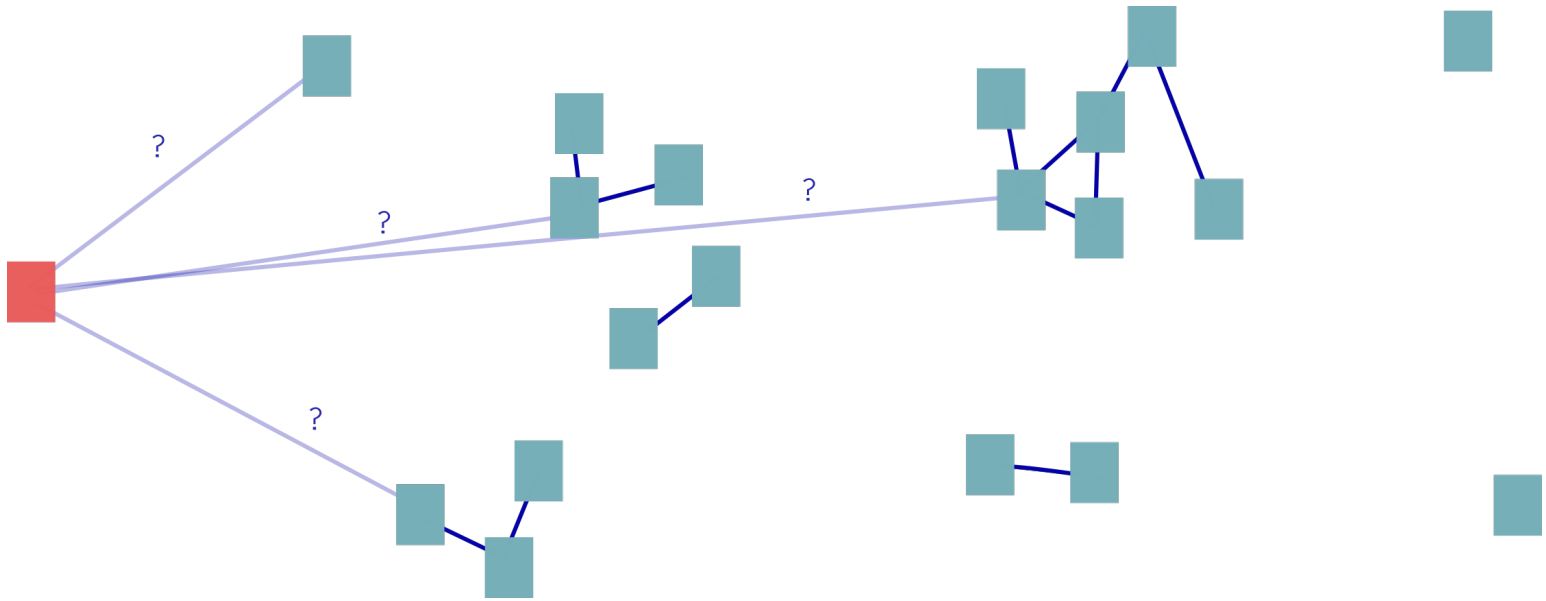
Generate pairs of documents whose sketches have at least one component in common:

GENPAIRS(Document set: $\{d_1, \ldots, d_N\}$ with sketches $\psi(d_h)$)

```
1 for k=1,...,200 do
2 |   Generate all pairs ⟨x_h^{π_k}, d_h⟩ (h = 1, ..., N)
3 |   Sort pairs on first component
4 |   for each block in sorted list with same first component do
5 |   |   return all pairs (d_h, d_{h'}) of second components
6 |   end
7 end
```

- Not counting line 5 the complexity is $O(N \log N)$
- Line 5 can still generate a large number of pairs (think of rather common shingles, e.g. “this is not a”)
- The same pair $(d_h, d_{h'})$ may be returned for different values of k.

Is a newly crawled page a near duplicate of a document already in the index?



- Compute sketch $\psi(d)$ of new document
- For each component x^{π_k} of $\psi(d)$: retrieve the documents d' that have the same value for x^{π_k} in their sketch as $\psi(d)$
- Estimate $J(S(d), S(d'))$ based on sketches $\psi(d), \psi(d')$