Crawling

collect + organize web pages.

HW3: topic-focused crawl in teams of 3

worked ibw

team → Dropbox

Northeastern University
College of Computer and Information Science

CS6200: Information Retrieval

Slides by: Jesse Anderton

Motivating Problem

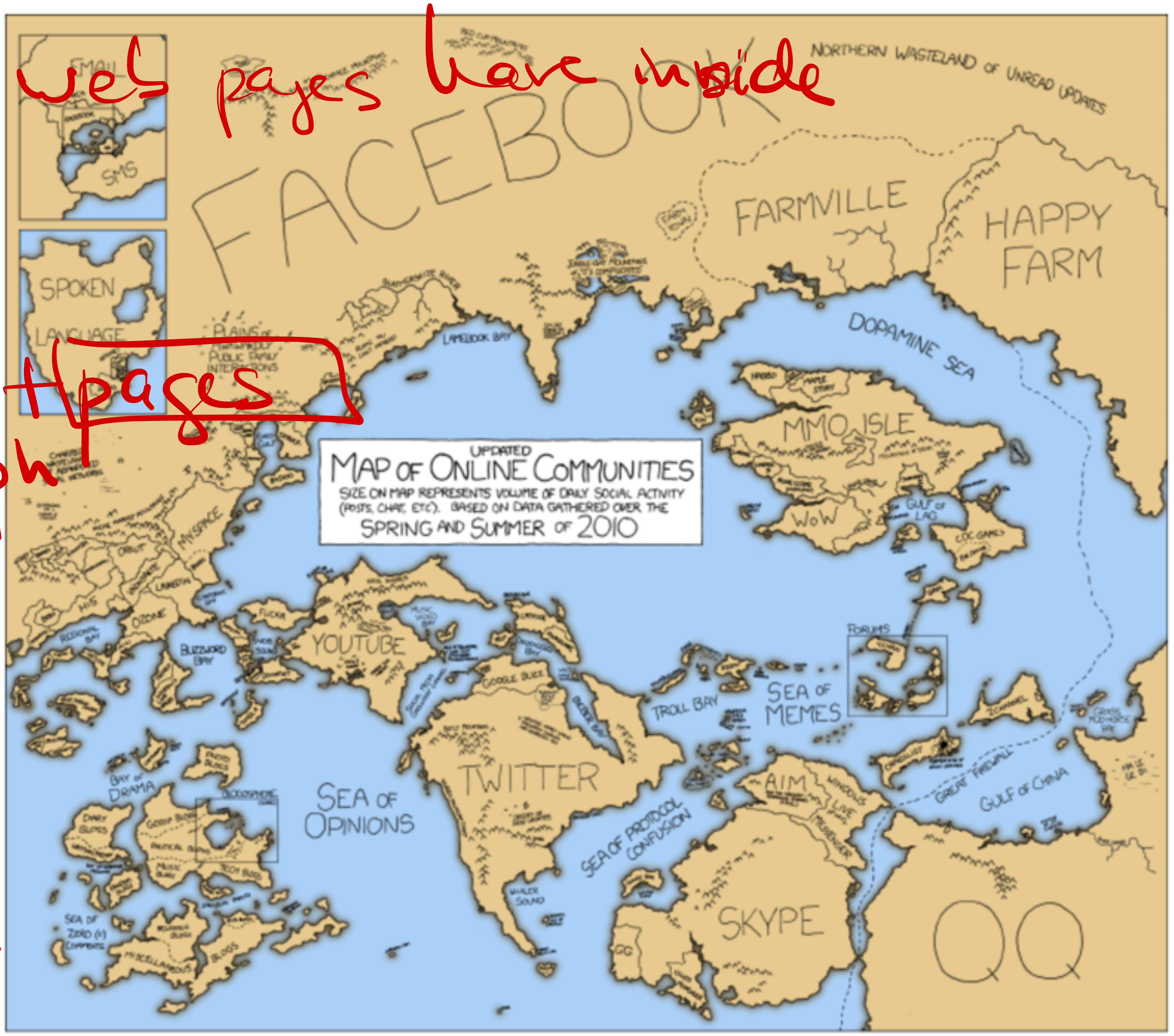
- we dont know what web pages have inside

Internet crawling is discovering web content and downloading it to add to your index.

- think of web link + pages as a graph

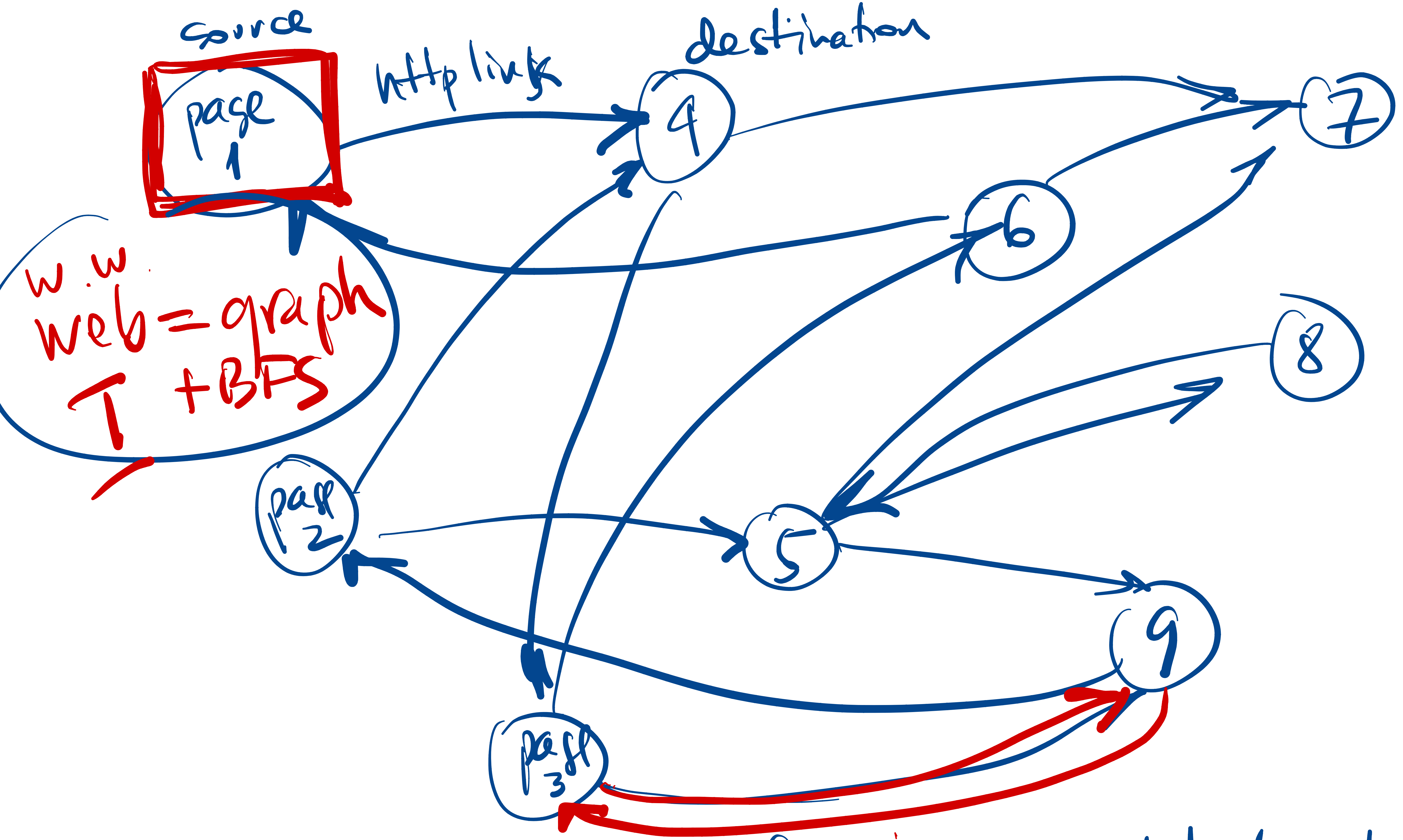
This is a technically complex, yet often overlooked aspect of search engines. “Breadth-first search from facebook.com” doesn’t begin to describe it.

crawler web = graph traversal/discovery



MAP OF ONLINE COMMUNITIES
UPDATED
SIZE ON MAP REPRESENTS VOLUME OF DAILY SOCIAL ACTIVITY (POSTS, CHAT, ETC). BASED ON DATA GATHERED OVER THE SPRING AND SUMMER OF 2010
NORTHERN WASTELAND OF UNREAD UPDATES
FACEBOOK
FARMVILLE
HAPPY FARM
EMAIL
SMS
SPOKEN LANGUAGE
LAMEBOOK BAY
DOPAMINE SEA
MMO ISLE
WoW
MYSPACE
YOUTUBE
TWITTER
SEA OF OPINIONS
TROLL BAY
SEA OF MEMES
FORUMS
SEA OF PROTOCOL CONFUSION
AIM
SKYPE
QQ
GULF OF CHINA
BAY OF DRAMA

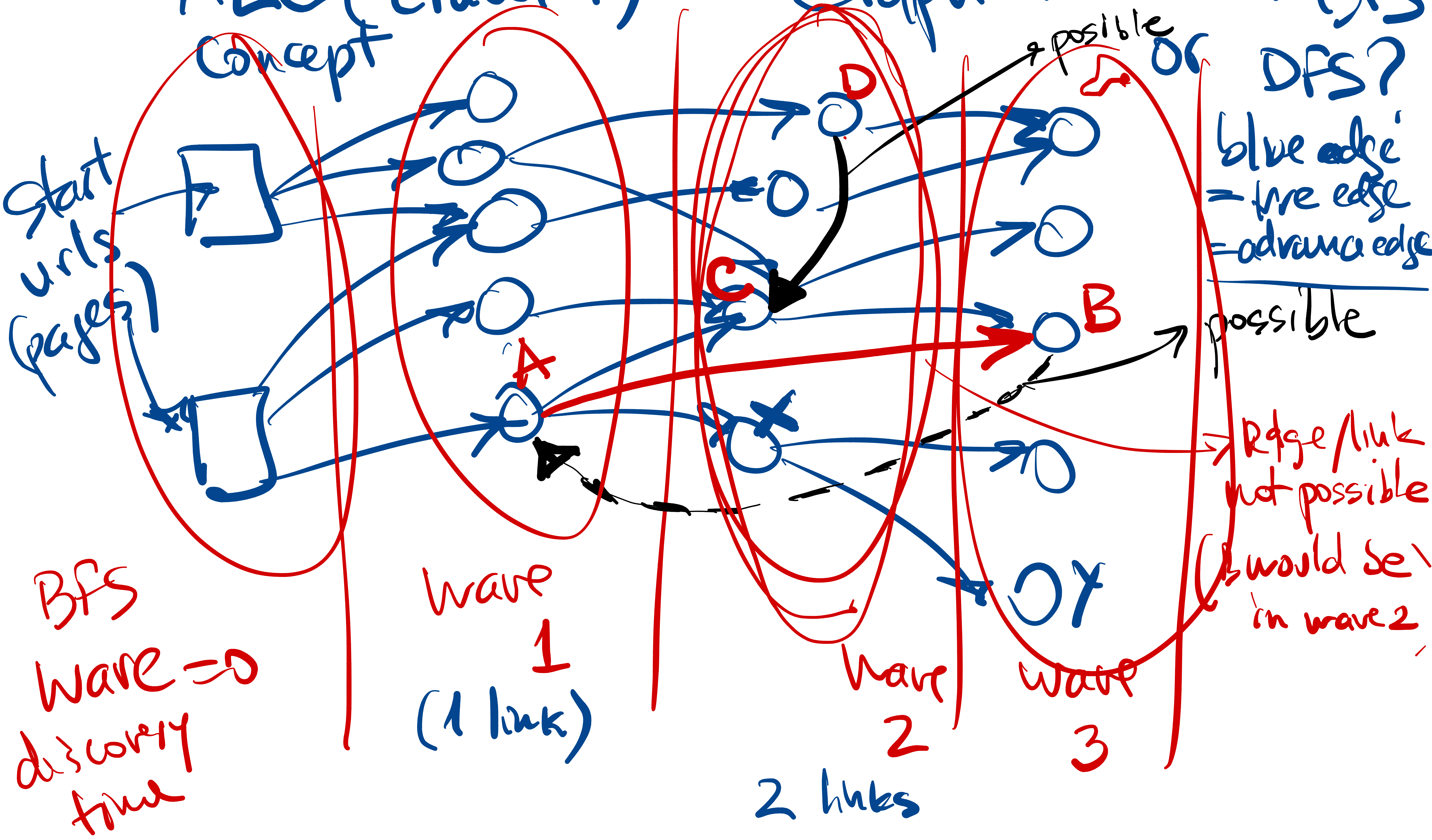
http://xkcd.com/802/



edges = http links found inside html content.

directed graph. Can we have cycles? Yes.

ALG (crawler) = Graph traversal BFS or DFS?



BFS

```
Queue = {start nodes}
while (condition ?)
    x = dequeue(Q)
    process(x)
    for each link x → y ?
        if (y = new) ? enqueue(y)
```

??

Queue FIFO → →

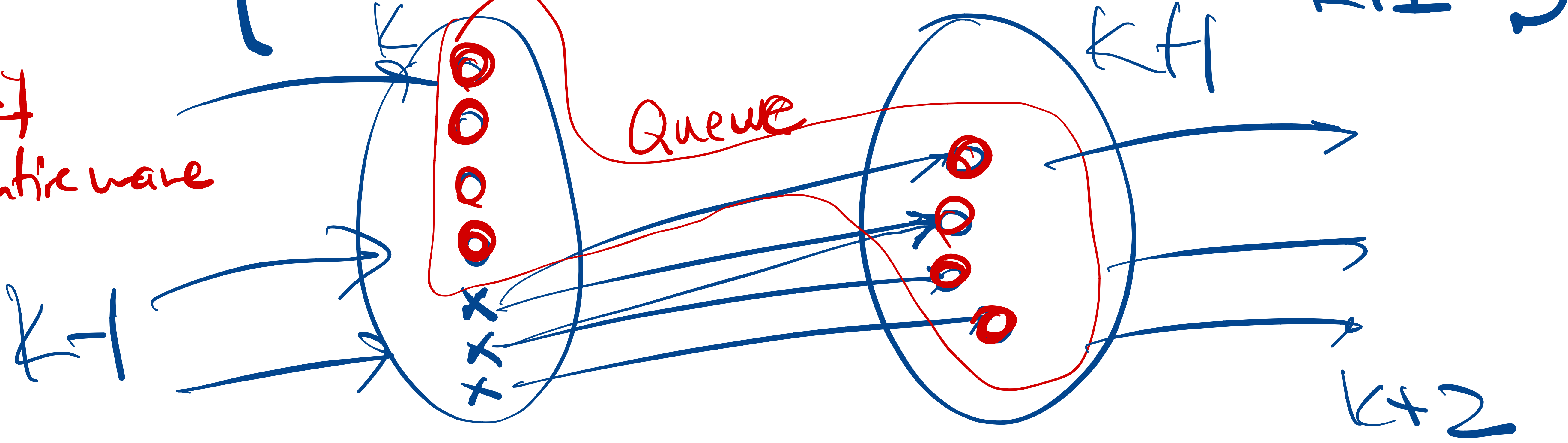
(at certain times) Queue = complete wave

claim

BFS Queue = { nodes not yet processed from wave k + nodes added from wave k+1 }

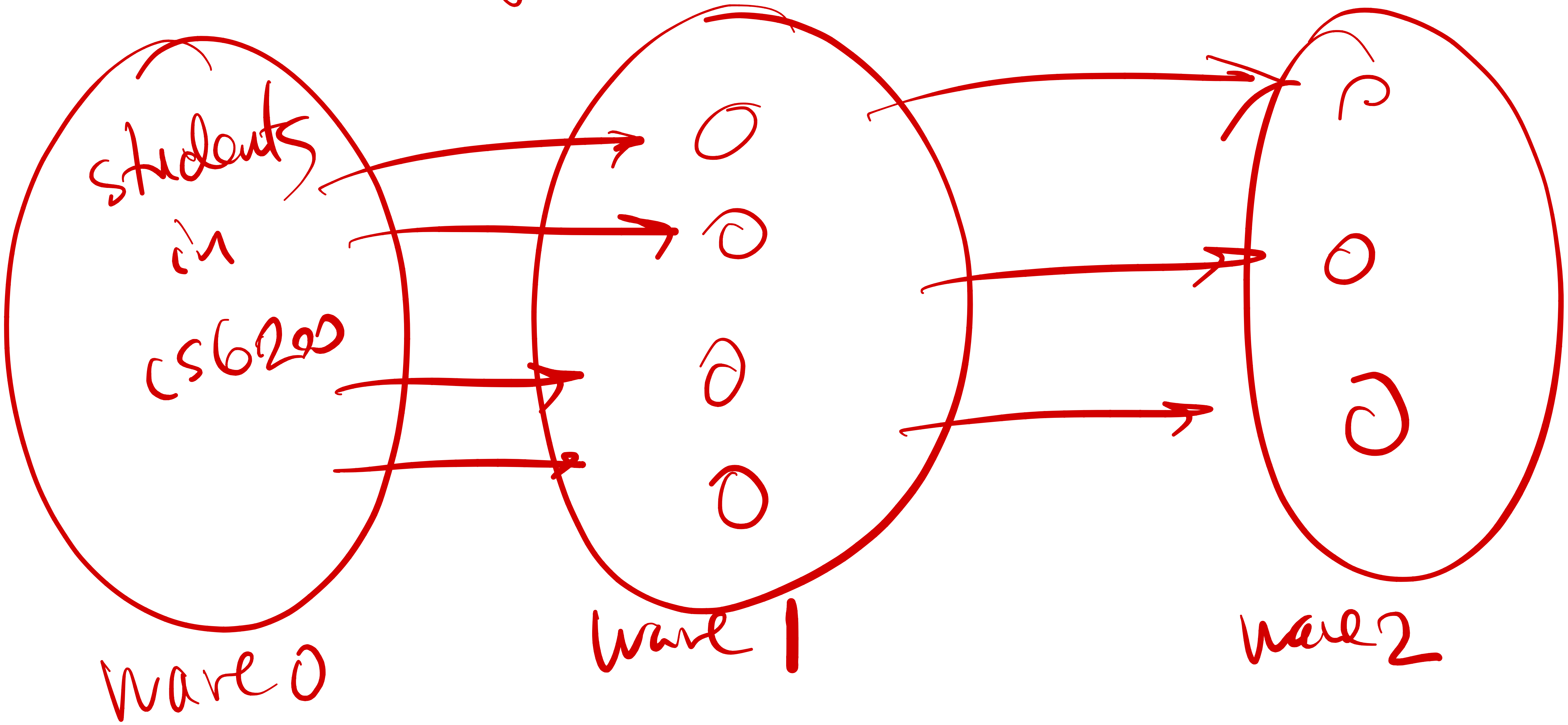
occasionally

Queue = entire wave



exponential increase in wave size?

facebook graph / friends



links both ways? yes

calendar?
"next month" link

1) nodes = web pages.
→ html content
parse,
store;
start urls = topical

Coverage

The first goal of an Internet crawler is to provide adequate coverage. *Coverage* is the fraction of available content you've crawled.

grand topic = "catholic church"
start { URL1, URL2, URL3, URL4

Challenges here include:

- Discovering new pages and web sites as they appear online.
- Duplicate site detection, so you don't waste time re-crawling content you already have.

calendar? (not adversarial)

- Avoiding *spider traps* – configurations of links that would cause a naive crawler to make an infinite series of requests.

adversarial spam/bots/spyware

Freshness

Coverage is often at odds with freshness. *Freshness* is the recency of the content in your index. If a page you've already crawled changes, you'd like to re-index it.

– relevant for recent/current content

– not so relevant for static/settled content

Freshness challenges include:

- Making sure your search engine provides good results for breaking news.
- Identifying the pages or sites which tend to be updated often.
- Balancing your limited crawling resources between new sites (coverage) and updated sites (freshness).

· crawler demand data >>> User demand.

Politeness

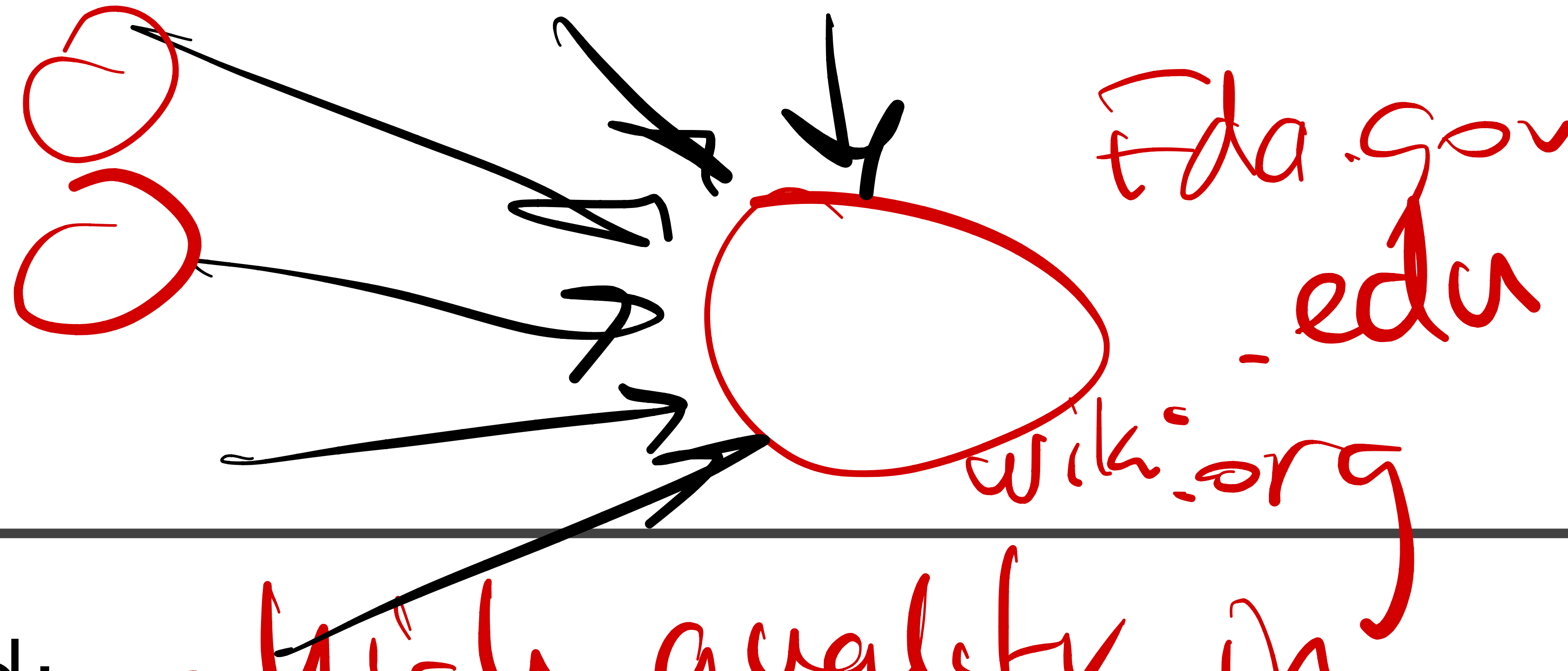
→ load on servers

Crawling the web consumes resources on the servers we're visiting. *Politeness* is a set of policies a well-behaved crawler should obey in order to be respectful of those resources.

- Requests to the same domain should be made with a reasonable delay. 1 sec / domain requested 0.2 sec for all domains
- The total bandwidth consumed from a single site should be limited.
- Site owners' preferences, expressed by files such as robots.txt, should be respected.

sitemap

And more…



Aside from these concerns, a good crawler should:

high quality in general

- Focus on crawling high-quality web sites.

quality = relevance for my topic

match (content, topic)

- Be distributed and scalable, and make efficient use of server resources.
- Crawl web sites from a geographically-close data center (when possible).
- Be extensible, so it can handle different protocols and web content types appropriately.

HTTP Crawling

Web crawling requires attending to many details. DNS responses should be cached, HTTP HEAD requests should generally be sent before GET requests, and so on.

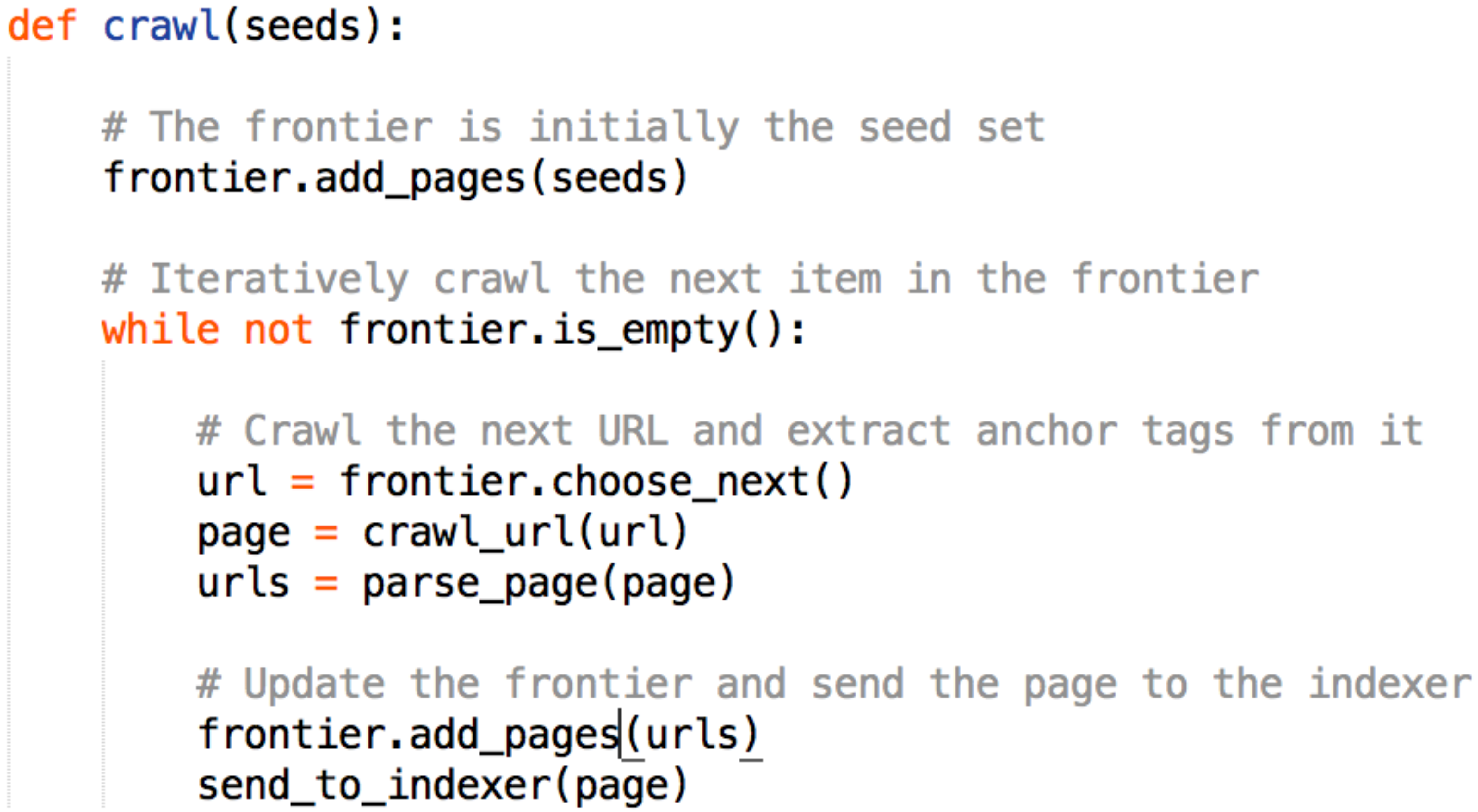
Extracting and normalizing URLs is important, because it dramatically affects your coverage and the time wasted on crawling, indexing, and ultimately retrieving duplicate content.

A Basic Crawler

A crawler maintains a *frontier* – a collection of pages to be crawled – and iteratively selects and crawls pages from it.

- The frontier is initialized with a list of *seed pages*.
- The next page is selected carefully, for politeness and performance reasons.
- New URLs are processed and filtered before being added to the frontier.

We will cover these details in subsequent sessions.



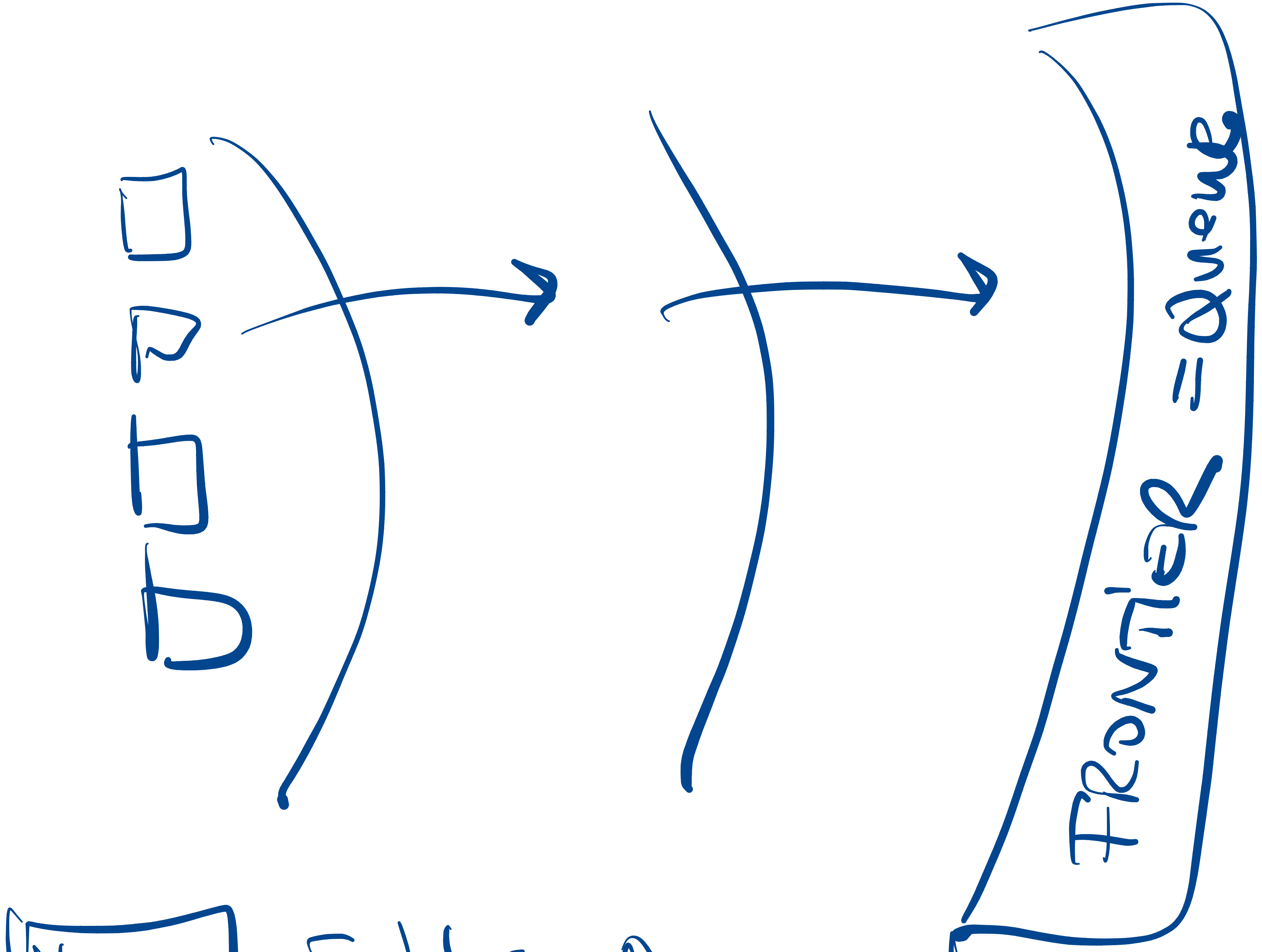
```
def crawl(seeds):

    # The frontier is initially the seed set
    frontier.add_pages(seeds)

    # Iteratively crawl the next item in the frontier
    while not frontier.is_empty():

        # Crawl the next URL and extract anchor tags from it
        url = frontier.choose_next()
        page = crawl_url(url)
        urls = parse_page(page)

        # Update the frontier and send the page to the indexer
        frontier.add_pages(urls)
        send_to_indexer(page)
```



theory: Frontier = Queue contains wave k (partial) + wave k+1 (partial)

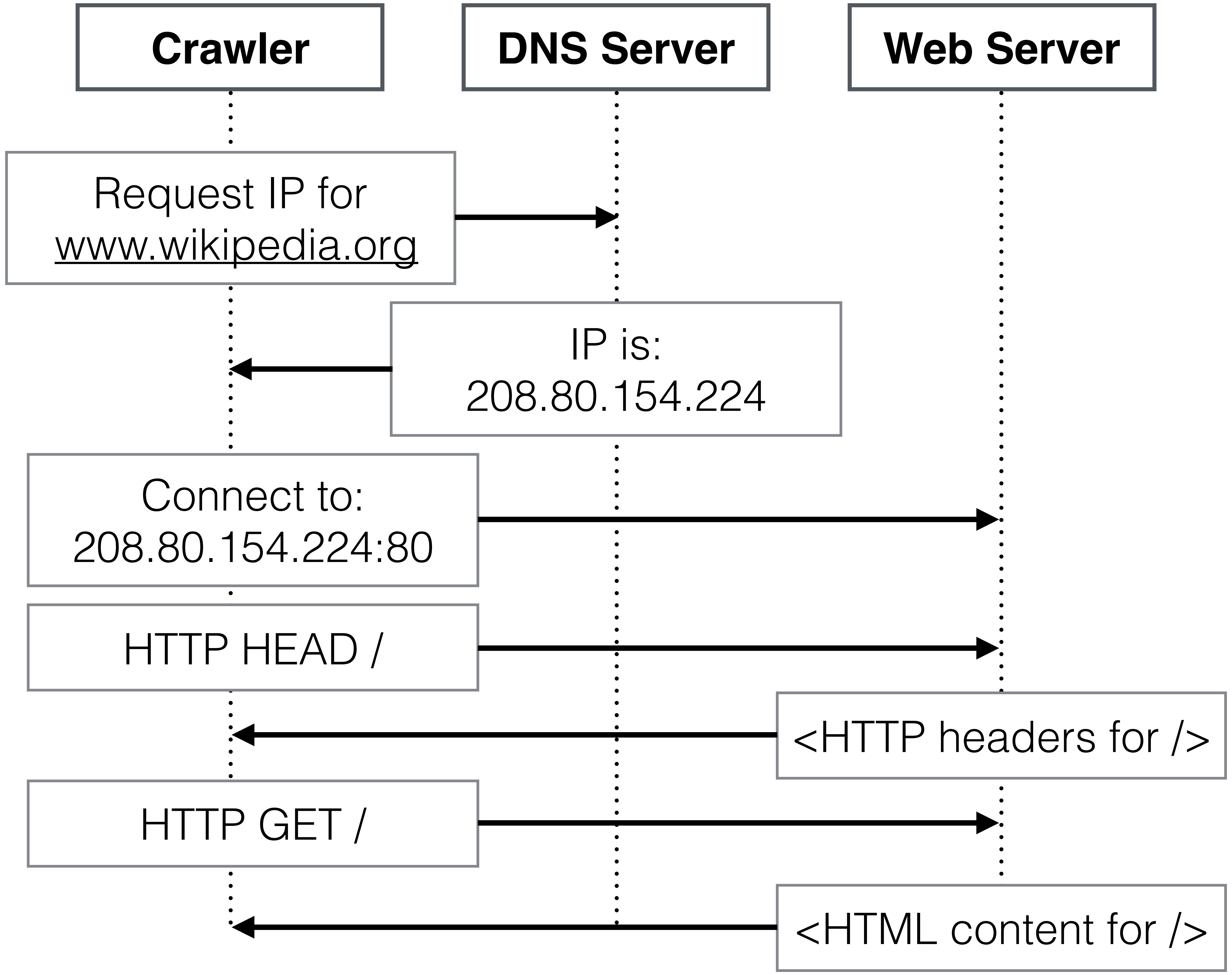
practice: Frontier Management

think of BFS-queue = FRONTier

HTTP Fetching

Requesting and downloading a URL involves several steps.

1. A DNS server is asked to translate the domain into an IP address.
2. (optional) An HTTP HEAD request is made at the IP address to determine the page type, and whether the page contents have changed since the last crawl.
3. An HTTP GET request is made to retrieve the new page contents.



Request process for http://www.wikipedia.org/

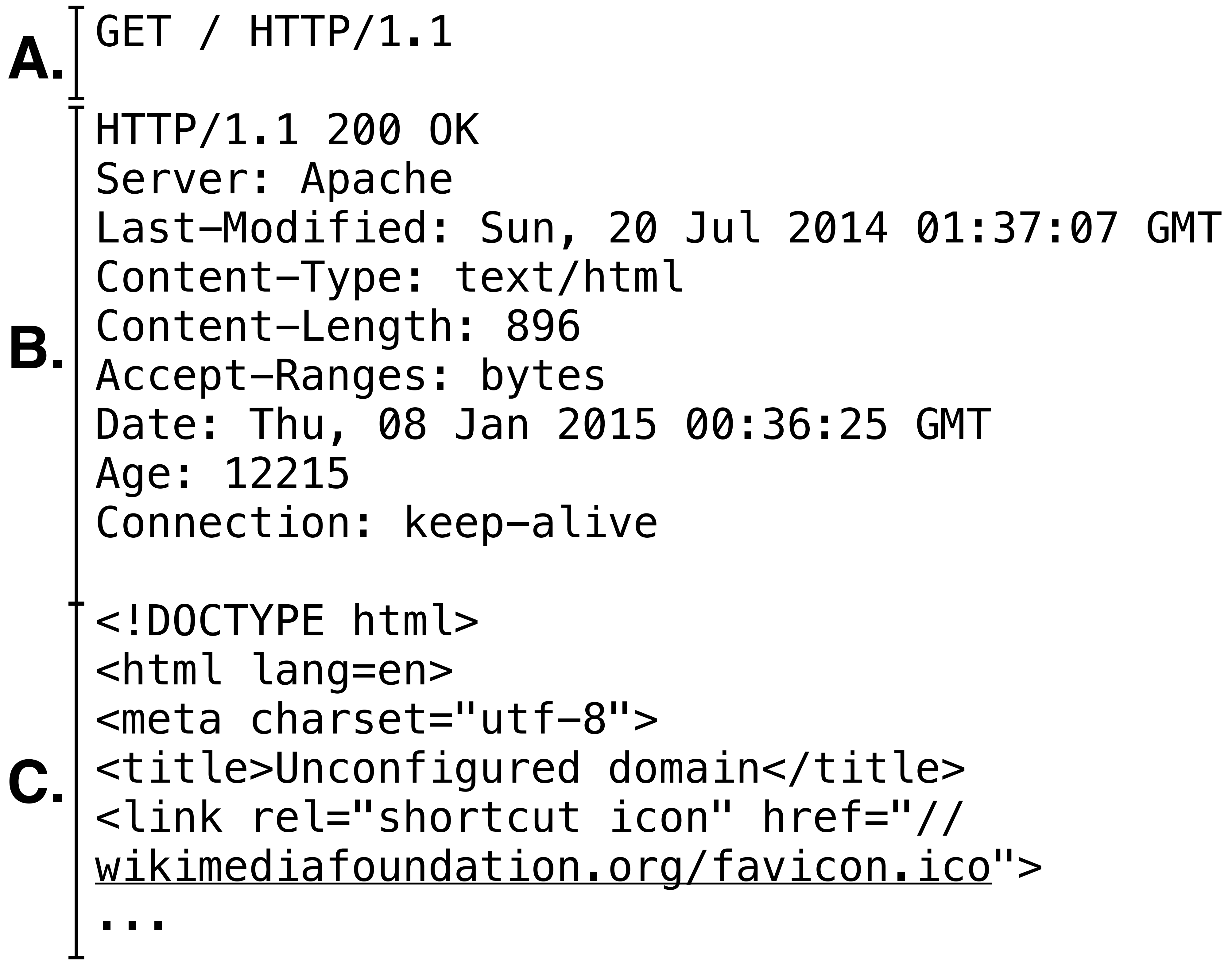
HTTP Requests

The HTTP request and response take the following form:

A. HTTP Request:
   <method> <url> <HTTP version>
   [<optional headers>]

B. Response Status and Headers:
   <HTTP version> <code> <status>
   [<headers>]

C. Response Body



```
GET / HTTP/1.1

HTTP/1.1 200 OK
Server: Apache
Last-Modified: Sun, 20 Jul 2014 01:37:07 GMT
Content-Type: text/html
Content-Length: 896
Accept-Ranges: bytes
Date: Thu, 08 Jan 2015 00:36:25 GMT
Age: 12215
Connection: keep-alive

<!DOCTYPE html>
<html lang=en>
<meta charset="utf-8">
<title>Unconfigured domain</title>
<link rel="shortcut icon" href="//
wikimediafoundation.org/favicon.ico">
...
```

HTTP Request and Response

URL Extraction

→ library

Downloaded files must be parsed according to their content type (usually available in the Content-Type header), and URLs extracted for adding to the frontier.

HTML documents in the wild often have formatting errors which the parser must address. Other document formats have their own issues. URLs may be embedded in PDFs, Word documents, etc.

Many URLs are missed, especially due to dynamic URL schemes and web pages generated by JavaScript and AJAX calls. This is part of the so-called “dark web.”

URL = page ID

share/library

→ consistent IDs

super important in team

URL Canonicalization

Many possible URLs can refer to the same resource. It's important for the crawler (and index!) to use a canonical, or normalized, version of the URLs to avoid repeated requests.

Many rules have been used; some are guaranteed to only rewrite URLs to refer to the same resource, and others can make mistakes.

It can also be worthwhile to create specific normalization rules for important web domains, e.g. by encoding which URL parameters result in different web content.

http://example.com/some/../folder?id=1#anchor

↓

http://example.com/some/../folder

↓

http://www.example.com/folder

Conversion to Canonical URL

→ ignore non-text

parse/clean web content ⇒ HARD/library

Rules for Canonicalization

Here are a few possible URL canonicalization rules.

Rule	Safe?	Example
Remove default port	Always	http://example.com:80 → http://example.com
Decoding octets for unreserved characters	Always	http://example.com/%7Ehome → http://example.com/~home
Remove . and ..	Usually	http://example.com/a/./b/../c → http://example.com/a/c
Force trailing slash for directories	Usually	http://example.com/a/b → http://example.com/a/b/
Remove default index pages	Sometimes	http://example.com/index.html → http://example.com
Removing the fragment	Sometimes	http://example.com/a#b/c → http://example.com/a

Duplicate Detection

Detecting near-duplicate page content is important for conserving limited crawling and indexing resources, and for making sure that search result lists contain meaningfully-distinct results.

Most approaches are based on hashing algorithms for fast but approximate similarity comparisons.

These techniques can also be readily adapted to detect plagiarism and copyright violations in plain text and in source code. In the latter case, document features often include syntactic parse trees rather than literal words.

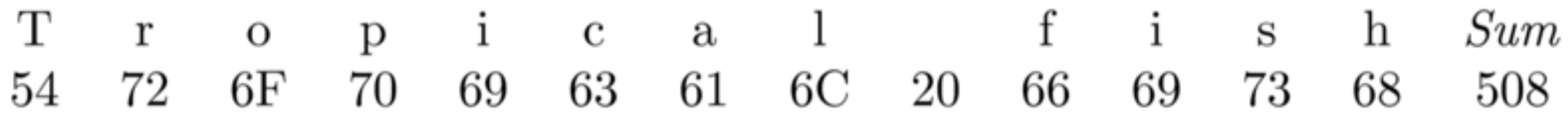
Page De-duplication

There are many duplicate or near-duplicate pages on the Internet: in a large crawl, roughly 30% of pages duplicate content found in the other 70%.

This happens for many reasons: copies, mirrors, versioning, plagiarism, spam, etc.

Exact duplicate detection is straightforward, relying on *checksums* for rapid detection.

A simple checksum: sum of bytes



T	r	o	p	i	c	a	l		f	i	s	h	*Sum*
54	72	6F	70	69	63	61	6C	20	66	69	73	68	508

Selected Checksum Types

Type	Example	Goal
Checksum	MD5	Duplicate detection
Error-correcting Checksum	Cyclic redundancy checks	Data verification and correction
Cryptographic Checksum	SHA-512	Non-reversible data identifiers
Hash Function	Jenkins hash functions	Hash table keys

Detecting Near-Duplicates

Detecting near-duplicates is much harder, especially with performance constraints.

A pairwise content comparison requires $O(n)$ comparisons to find matches for a single document, or $O(n^2)$ to find matches for all pairs.

We will explore two approaches here:

- Fingerprint methods generate a smaller document description which can be used for faster approximate comparison based on ngram overlap.

- Similarity hashing efficiently calculates approximate cosine similarity between documents.

Fingerprint Calculation

Fingerprints based on n-grams can be calculated and used as follows.

1. A document is converted into a sequence of words; all punctuation and formatting is removed.

2. Words are grouped into (possibly overlapping) n-grams, for some n.

3. A subset of the n-grams are selected to represent the document.

4. The selected n-grams are hashed, and the resulting hashes stored in an inverted index.

5. Documents are compared based on the number of overlapping n-grams.

Fingerprint Example

1. Original text

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

2. Overlapping trigrams

tropical fish include, fish include fish, include fish found, fish found in, found in tropical, in tropical environments, tropical environments around, environments around the, around the world, the world including, world including both, including both freshwater, both freshwater and, freshwater and salt, and salt water, salt water species

3. Hashed trigrams

938 664 463 822 492 798 78 969 143 236 913 908 694 553 870 779

4. Selected trigrams (using *0 mod 4*)

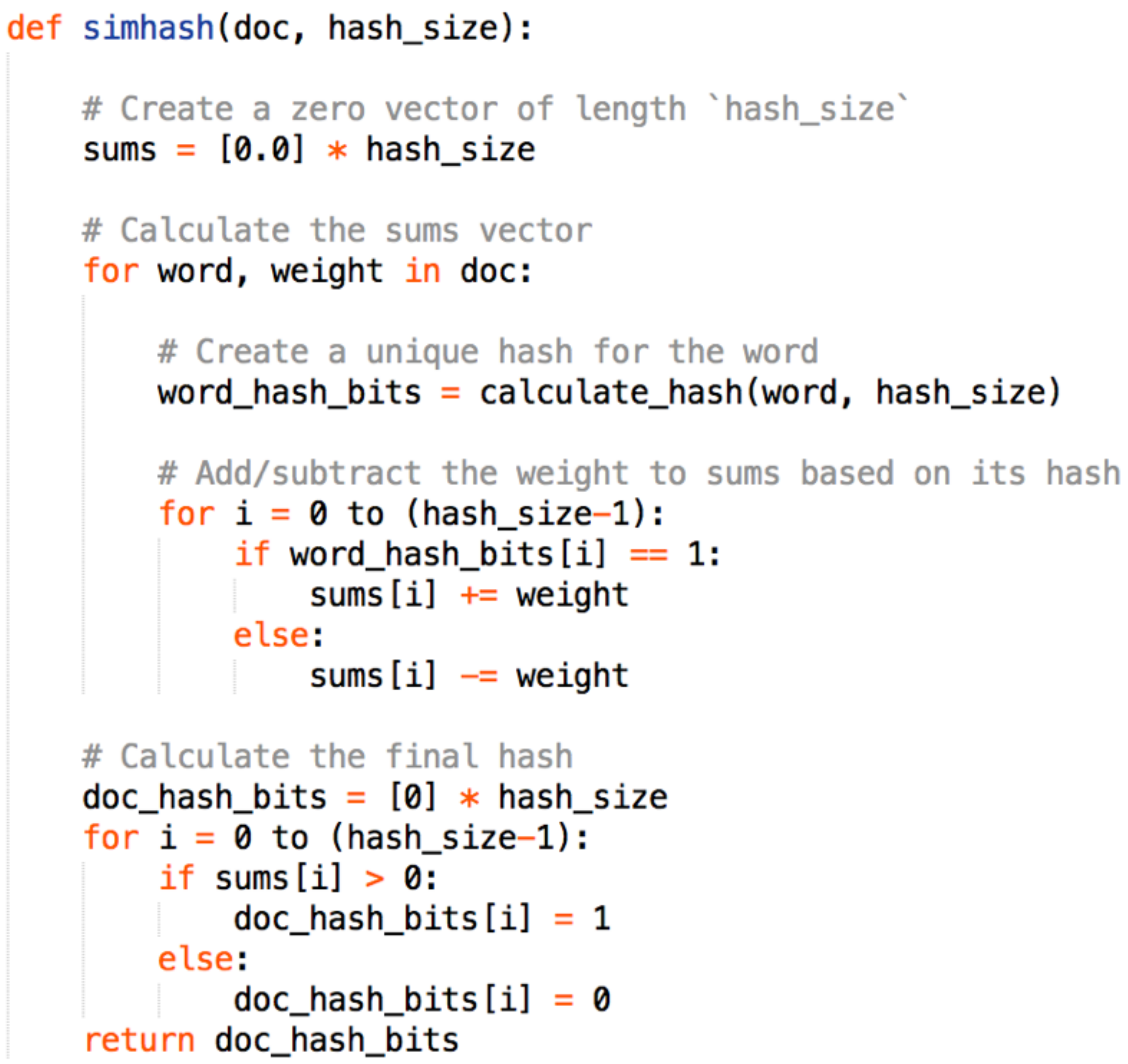
664 492 236 908

Similarity Hashing

Word-based comparisons are more effective at finding near-duplicates than comparing n-grams, but efficiency is a problem.

The simhash algorithm performs word-based comparisons with a hashing approach similar to n-gram fingerprints. It can be run over any weighted document features.

The cosine similarity between two documents is proportional to the number of bits in common between their simhash fingerprints.



```
def simhash(doc, hash_size):

    # Create a zero vector of length `hash_size`
    sums = [0.0] * hash_size

    # Calculate the sums vector
    for word, weight in doc:

        # Create a unique hash for the word
        word_hash_bits = calculate_hash(word, hash_size)

        # Add/subtract the weight to sums based on its hash
        for i = 0 to (hash_size-1):
            if word_hash_bits[i] == 1:
                sums[i] += weight
            else:
                sums[i] -= weight

    # Calculate the final hash
    doc_hash_bits = [0] * hash_size
    for i = 0 to (hash_size-1):
        if sums[i] > 0:
            doc_hash_bits[i] = 1
        else:
            doc_hash_bits[i] = 0
    return doc_hash_bits
```

Similarity Hashing Example

1. Original text

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

2. Weighted features (word TF scores)

tropical: 2, fish: 2, include: 1, found: 1, environments: 1, around: 1, world: 1, including: 1, both: 1, freshwater: 1, salt: 1, water: 1, species: 1

3. 8-bit Word Hashes

tropical: 01100001, fish: 10101011, include: 11100110, found: 00011110,
environments: 00101101, around: 10001011, world: 00101010, including: 11000000,
both: 10101110, freshwater: 00111111, salt: 10110101, water: 00100101, species: 11101110

4. Summed feature weights

sums = [1, -5, 9, -9, 3, 1, 3, 3]

5. Document Hash

10101111

Politeness

It's important to remember that we are providing web site owners with a service, and to be mindful of the resources we consume by crawling their sites.

Following robots.txt and using sitemaps.xml can also help the crawler avoid pitfalls particular to the web site.

The need for politeness

Web crawlers are generally distributed and multithreaded, and are capable of taxing the capabilities of most web servers. This is particularly true of the crawlers for major businesses, such as Bing and Google.

In addition, some content should not be crawled at all.

- Some web content is considered private or under copyright, and its owners prefer that it not be crawled and indexed.

- Other URLs implement API calls and don’t lead to indexable content.

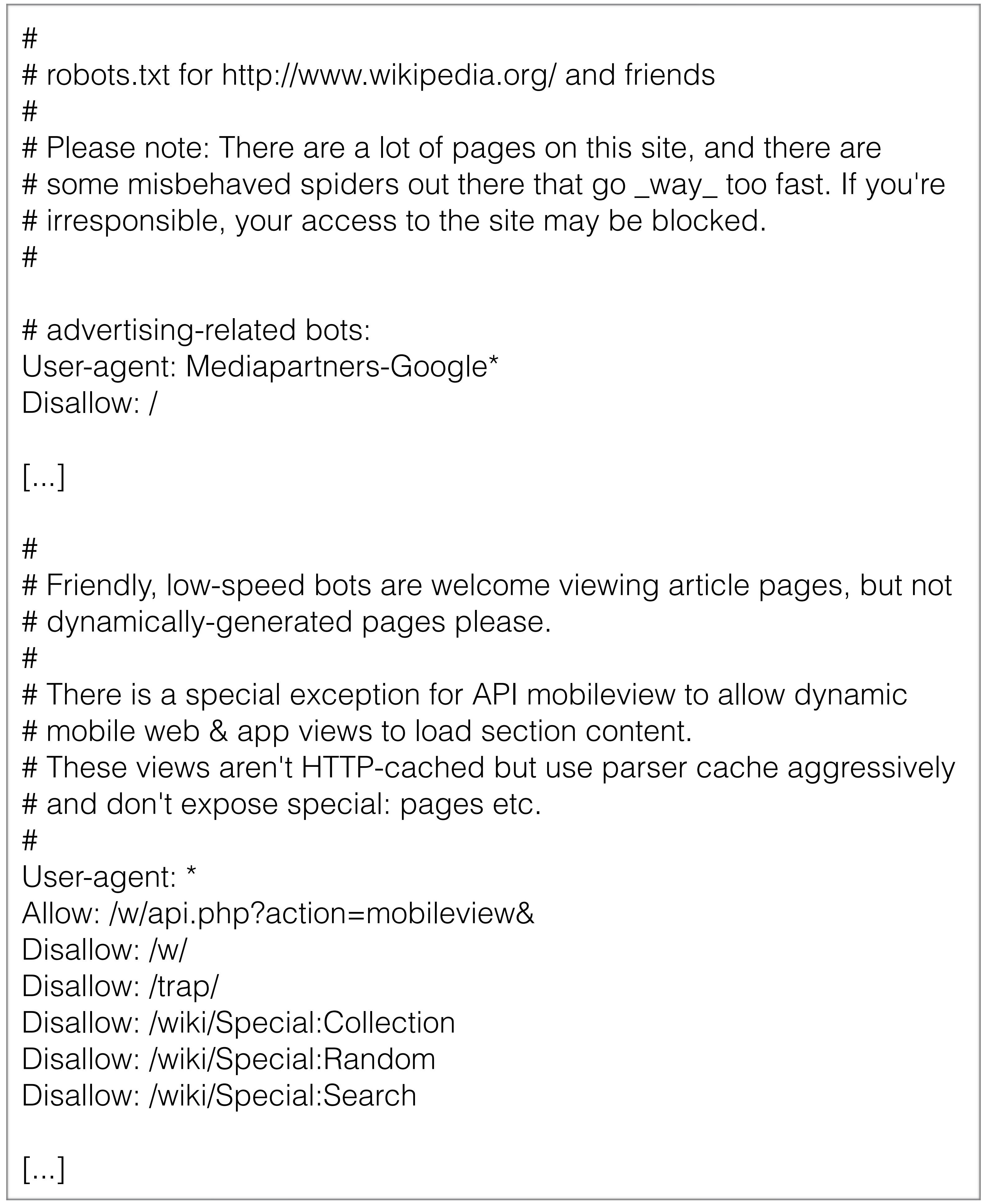
This can easily create conflict between search providers and the web sites they’re linking to. Politeness policies have been created as a way to mediate these issues.

Robots Exclusion Protocol

Web site administrators can express their crawling preferences by hosting a page at /robots.txt. Every crawler should honor these preferences.

These files indicate which files are permitted and disallowed for particular crawlers, identified by their user agents. They can also specify a preferred site mirror to crawl.

More recently, sites have also begun requesting crawl interval delays in robots.txt.



```
#
# robots.txt for http://www.wikipedia.org/ and friends
#
# Please note: There are a lot of pages on this site, and there are
# some misbehaved spiders out there that go _way_ too fast. If you're
# irresponsible, your access to the site may be blocked.
#

# advertising-related bots:
User-agent: Mediapartners-Google*
Disallow: /

[...]

#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically-generated pages please.
#
# There is a special exception for API mobileview to allow dynamic
# mobile web & app views to load section content.
# These views aren't HTTP-cached but use parser cache aggressively
# and don't expose special: pages etc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Disallow: /w/
Disallow: /trap/
Disallow: /wiki/Special:Collection
Disallow: /wiki/Special:Random
Disallow: /wiki/Special:Search

[...]
```

Portion of http://www.wikipedia.org/robots.txt

Request Intervals

It is very important to limit the rate of requests your crawler makes to the same domain. Too-frequent requests are the main way a crawler can harm a web site.

Typical crawler delays are in the range of 10-15 seconds per request. You should never crawl more than one page per second from the same domain. For large sites, it can take days or weeks to crawl the entire domain – this is preferable to overloading their site (and possibly getting your IP address blocked).

If the site’s robots.txt file has a Crawl-delay directive, it should be honored.

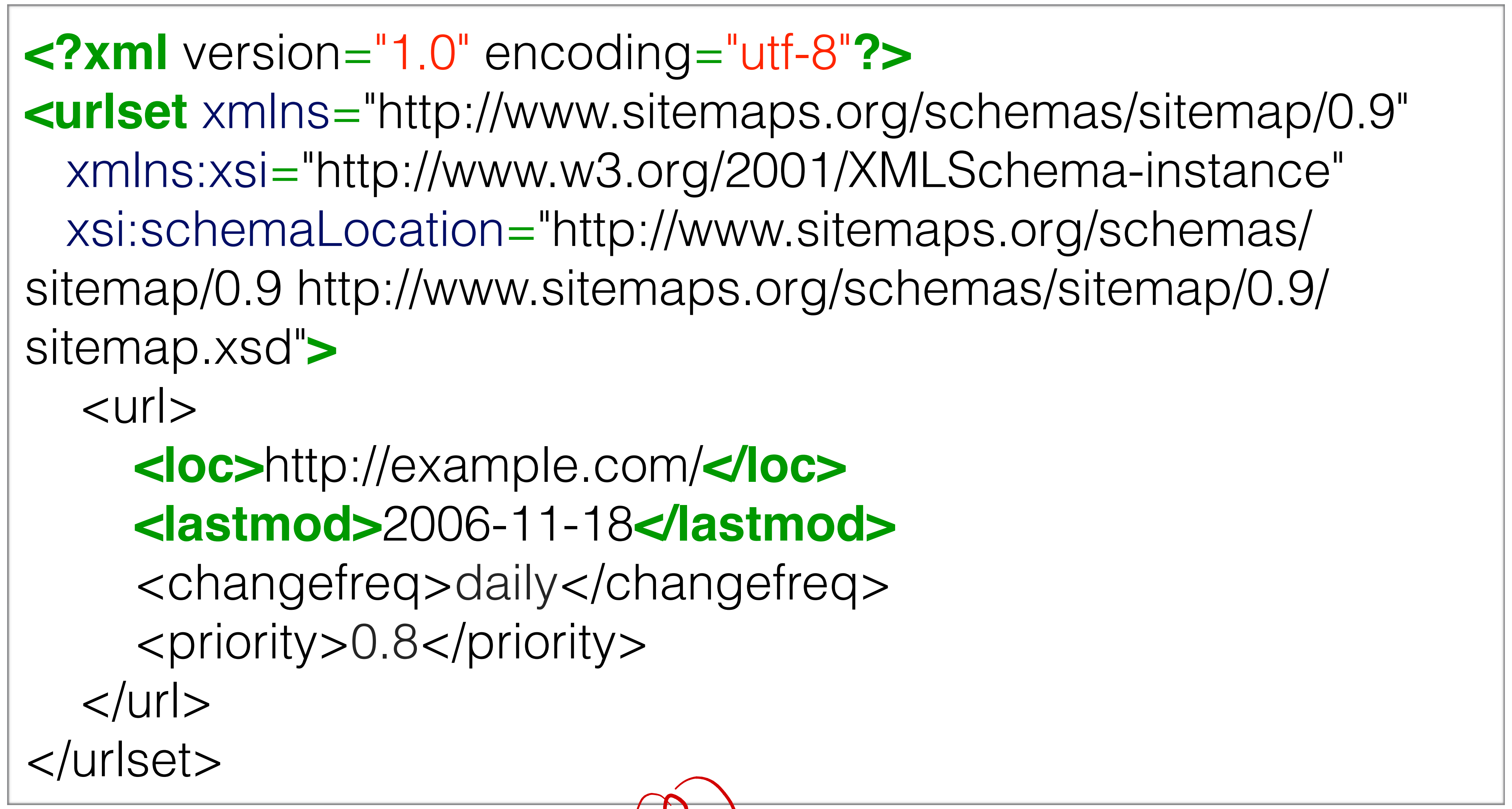
In a distributed crawler, all requests for the same domain are typically sent to the same crawler instance to easily throttle the rate of requests.

Sitemaps

In addition to robots.txt, which asks crawlers *not* to index certain content, a site can request that certain content be indexed by hosting a file at /sitemap.xml.

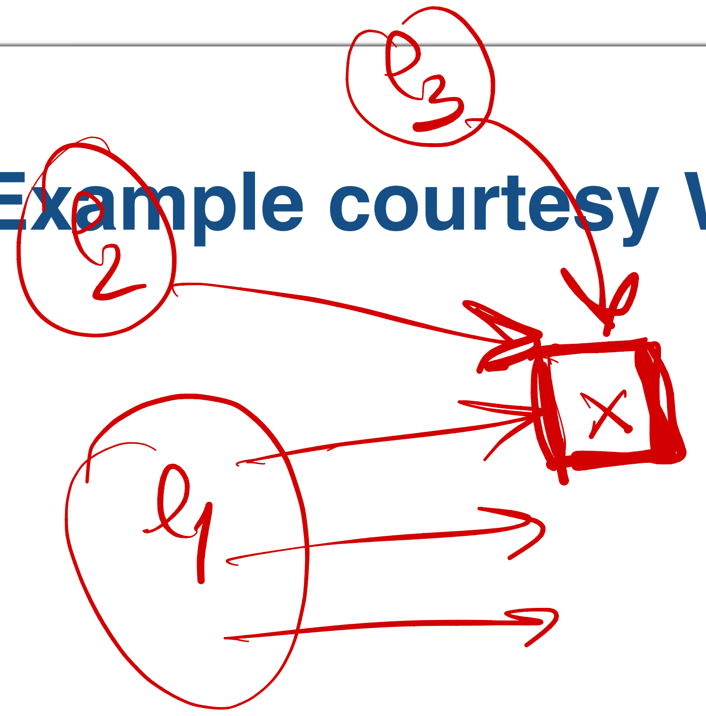
Sitemaps can direct your crawler toward the most important content on the site, indicate when it has changed, etc.

||| engineering + team + ES
retrieval / evaluation ?



```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.sitemaps.org/schemas/
sitemap/0.9 http://www.sitemaps.org/schemas/sitemap/0.9/
sitemap.xsd">
    <url>
        <loc>http://example.com/</loc>
        <lastmod>2006-11-18</lastmod>
        <changefreq>daily</changefreq>
        <priority>0.8</priority>
    </url>
</urlset>
```

Example courtesy Wikipedia



F = frontier = {start URLs} ①

set of URL_current C = dequeue(F, batch_size)

② A For each $l \in C$ (current set) – http head call ⇒ meta info

GET ⟹ – wget call ⇒ content page

✱ polite

B

CLEAN • Extract links from page l ⟹ canonical

$o_1, o_2, o_3, \dots o, \dots o_z$

• Analyze content → relevant?

→ new/diverse?

→ duplicate?

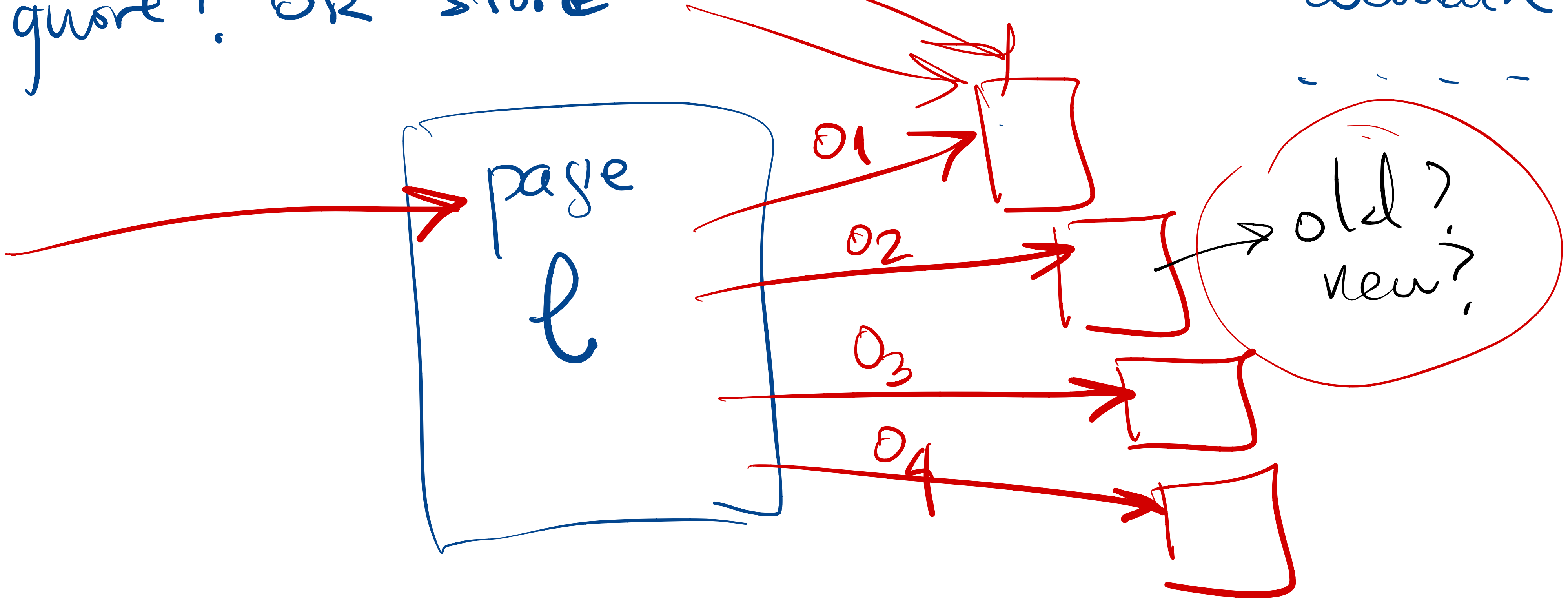
→ pagerank = ?

→ domain = ?

.... ?

ignore? OR store

eac(



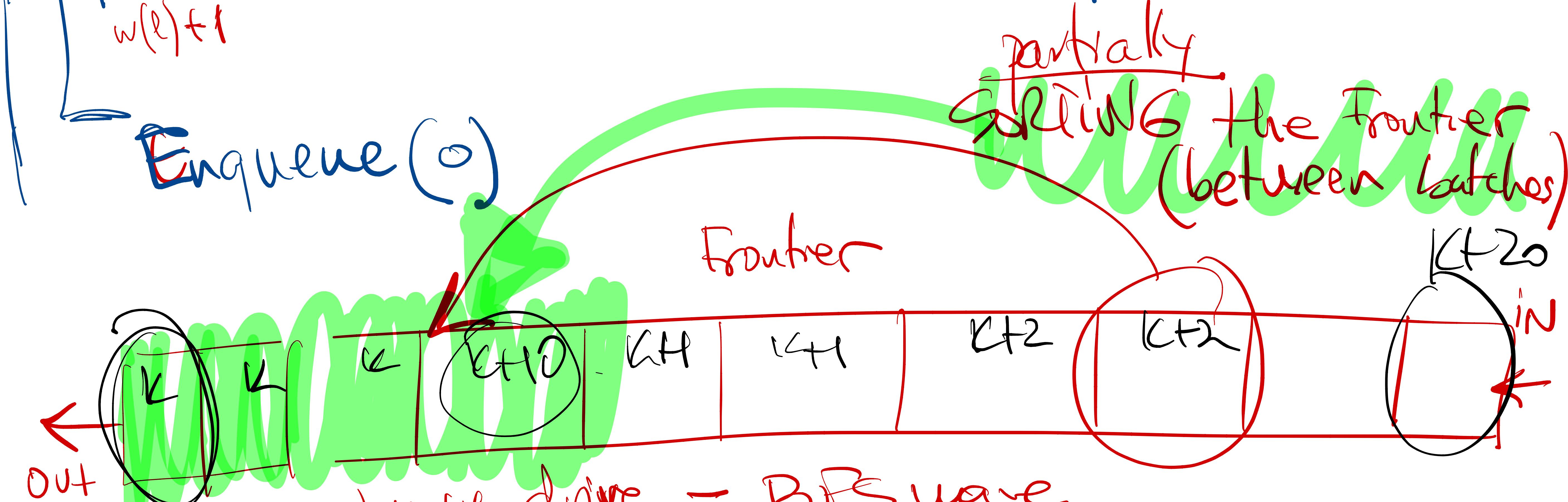
(C) STORE(e) links outgoing

- ~~content~~ text + URLs + everything we need to keep ⇒ ES

- foreach OF outgoing links ignore(old/seen)

0	BFS	incoming to o	URL(o) text/keywords/anchor	security	domain
Gure	wave	count	relevant?	filter	?
	w(e)+1				

Enqueue(o)



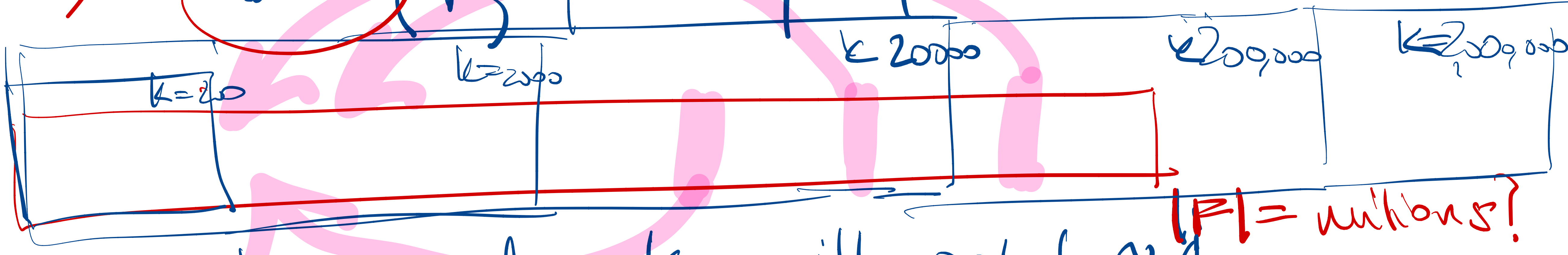
Score
- primary drive = BFS wave
- relevance
- importance

grand topic
seed URLs } → manual / keywords
pseudo

Ⓕ Management ~ mostly sorting

between batches (dequeue)

- update score(o)
- sort (F, partially top k) → complex sort on multiple criteria



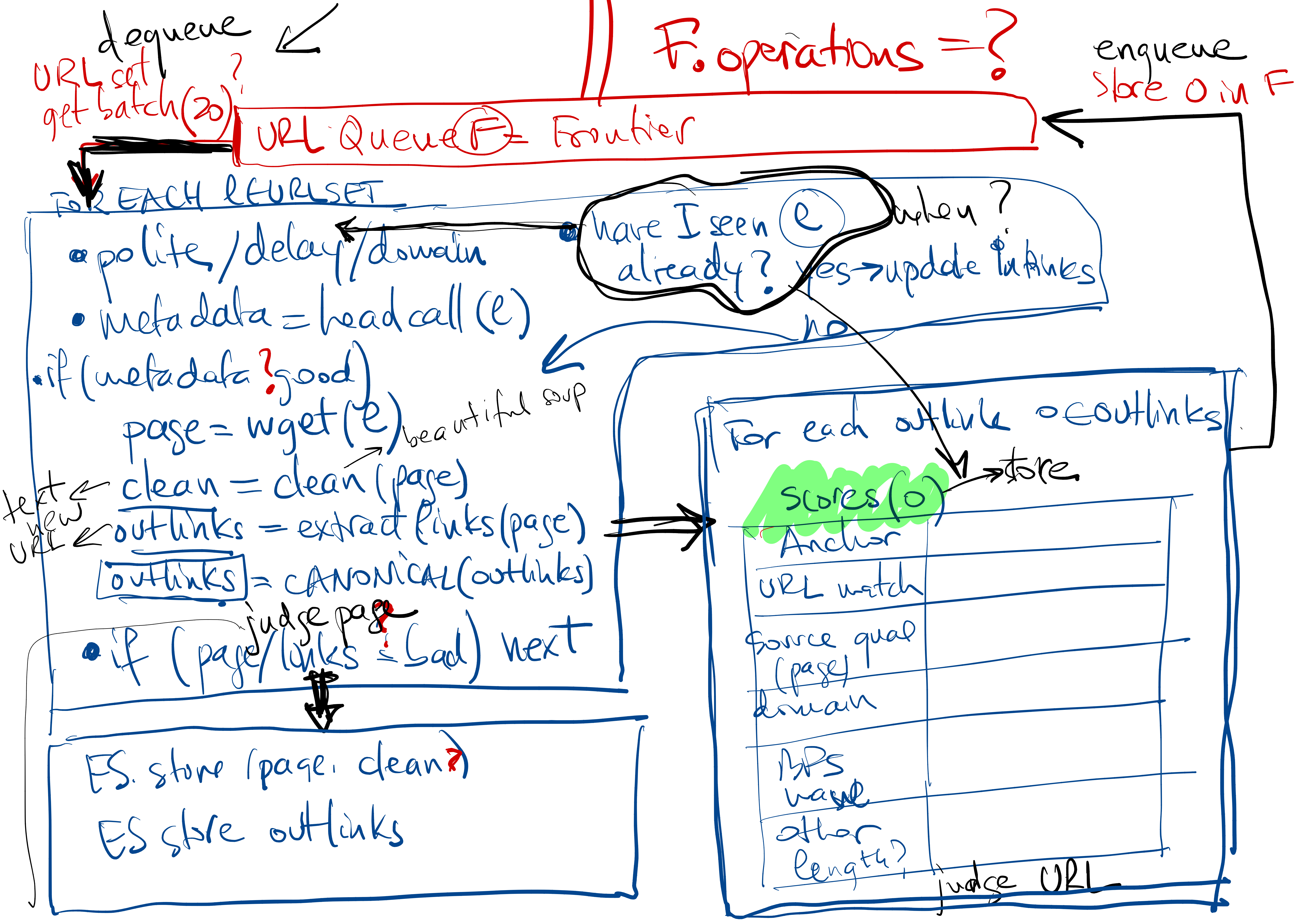
choose random k with probab ~ 1

- often k=100
- sometimes k=1000
- rarely k=10000 once/twice
- k=|F|

- before every dequeue (batch maybe)

↑

batch = #URLs processed before resorting F


dequeue
URL set
get batch(20) ?
F. operations = ?
enqueue
store o in F
URL Queue F = Frontier
FOR EACH e ∈ URLSET
• polite/delay/domain
• metadata = headcall(e)
• if (metadata ? good)
page = wget(e)
beautiful soup
clean = clean(page)
text
outlinks = extract links(page)
new URL
outlinks = CANONICAL(outlinks)
judge page
• if (page/links ? bad) next
ES. store (page, clean)
ES store outlinks
have I seen e already?
when?
yes → update inlinks
no
For each outlink o ∈ outlinks
scores(o)
store
Anchor
URL match
Source qual (page)
domain
BFS wave
other length?
judge URL

what makes a page good?

- content relevance / matching keywords
- coherent layout
- coherent text
- advertise / spam
- length
- domain
- recency (of this page)

- links-to-text ratio / #outgoing links
- # of incoming links / ~ pagerank
- navigation within the page