

# CS6322: Information Retrieval

## Sanda Harabagiu

### Lecture 9: Crawling and web indexes



# Today's lecture

---

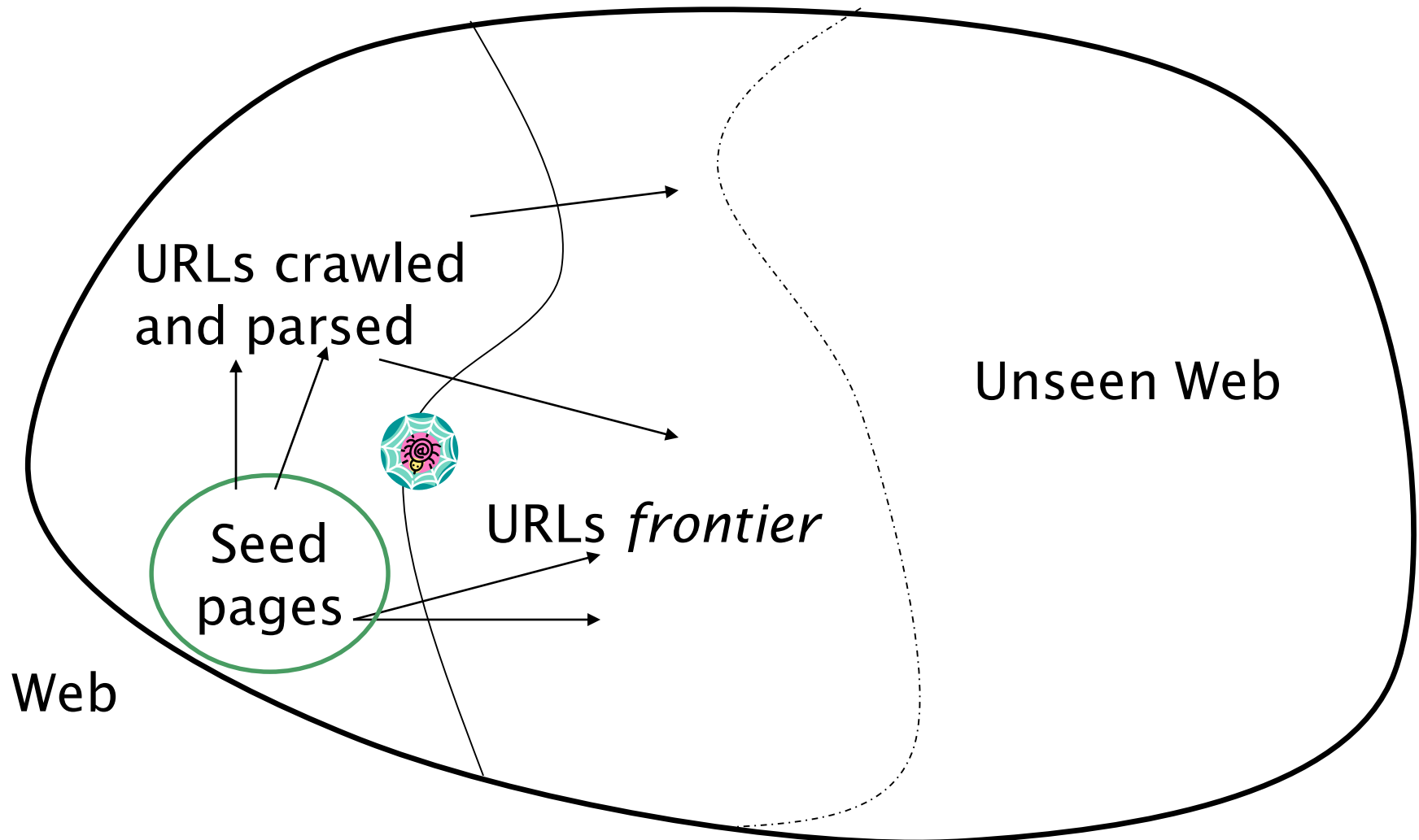
- Crawling
- Connectivity servers

# Basic crawler operation

---

- Begin with known “seed” URLs
- Fetch and parse them
  - Extract URLs they point to
  - Place the extracted URLs on a queue
- Fetch each URL on the queue and repeat

# Crawling picture



# Simple picture – complications

---

- Web crawling isn't feasible with one machine
  - All of the above steps distributed
- Malicious pages
  - Spam pages
  - Spider traps – incl dynamically generated
- Even non-malicious pages pose challenges
  - Latency/bandwidth to remote servers vary
  - Webmasters' stipulations
    - How “deep” should you crawl a site's URL hierarchy?
  - Site mirrors and duplicate pages
- Politeness – don't hit a server too often

# What any crawler *must* do

---

- Be Polite: Respect implicit and explicit politeness considerations
  - Only crawl allowed pages
  - Respect *robots.txt* (more on this shortly)
- Be Robust: Be immune to spider traps and other malicious behavior from web servers

# What any crawler *should* do

---

- Be capable of distributed operation: designed to run on multiple distributed machines
- Be scalable: designed to increase the crawl rate by adding more machines
- Performance/efficiency: permit full use of available processing and network resources

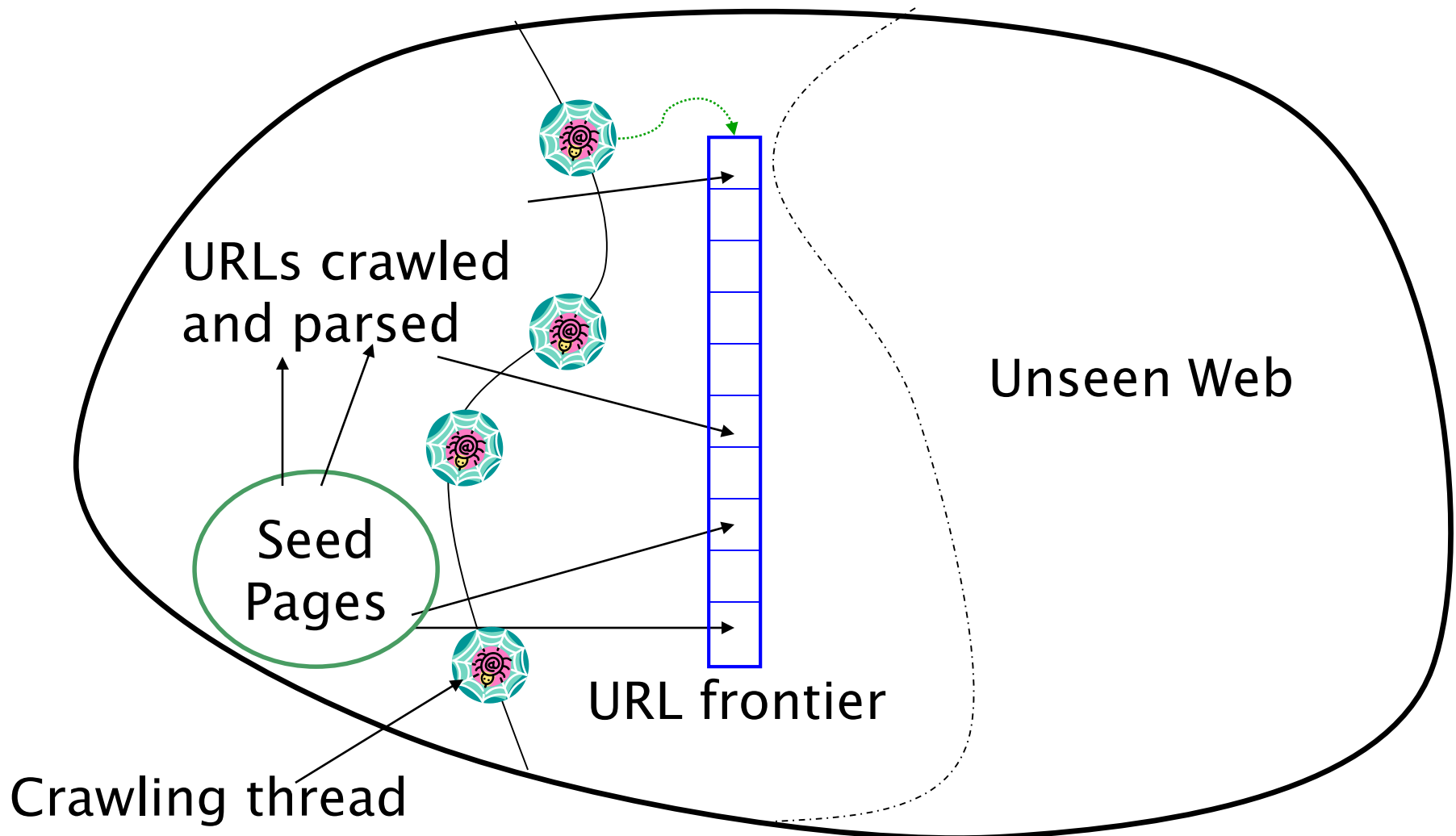
# What any crawler *should* do

---

- Fetch pages of “higher quality” first
- Continuous operation: Continue fetching fresh copies of a previously fetched page
- Extensible: Adapt to new data formats, protocols



# Updated crawling picture



# URL frontier

---

- Can include multiple pages from the same host
- Must avoid trying to fetch them all at the same time
- Must try to keep all crawling threads busy

# Explicit and implicit politeness

---

- Explicit politeness: specifications from webmasters on what portions of site can be crawled
  - robots.txt
- Implicit politeness: even with no specification, avoid hitting any site too often

# Robots.txt

---

- Protocol for giving spiders (“robots”) limited access to a website, originally from 1994
  - [www.robotstxt.org/wc/norobots.html](http://www.robotstxt.org/wc/norobots.html)
- Website announces its request on what can(not) be crawled
  - For a URL, create a file `URL/robots.txt`
  - This file specifies access restrictions

# Robots.txt example

---

- No robot should visit any URL starting with `"/yoursite/temp/"`, except the robot called `"searchengine"`:

```
User-agent: *
```

```
Disallow: /yoursite/temp/
```

```
User-agent: searchengine
```

```
Disallow:
```

# Processing steps in crawling

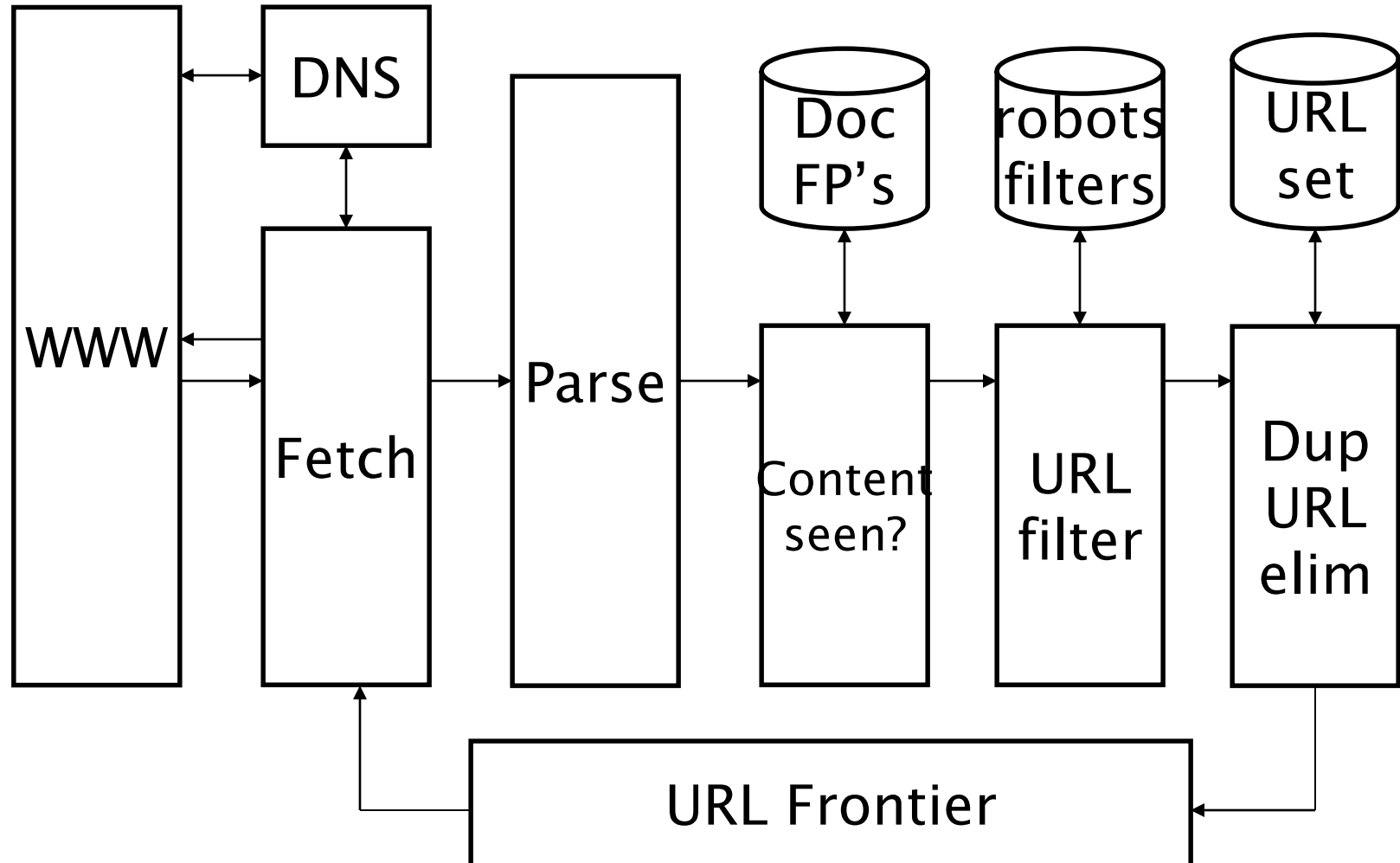
---

- Pick a URL from the frontier
- Fetch the document at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier (duplicate URL elimination)

← Which one?

E.g., only crawl .edu,  
obey robots.txt, etc.

# Basic crawl architecture



# DNS (Domain Name Server)

---

- A lookup service on the internet
  - Given a URL, retrieve its IP address
  - Service provided by a distributed set of servers – thus, lookup latencies can be high (even seconds)
- Common OS implementations of DNS lookup are *blocking*: only one outstanding request at a time
- Solutions
  - DNS caching
  - Batch DNS resolver – collects requests and sends them out together



# Parsing: URL normalization

---

- When a fetched document is parsed, some of the extracted links are *relative* URLs
- E.g., at [http://en.wikipedia.org/wiki/Main\\_Page](http://en.wikipedia.org/wiki/Main_Page)  
we have a relative link to  
/wiki/Wikipedia:General\_disclaimer which is the same  
as the absolute URL  
[http://en.wikipedia.org/wiki/Wikipedia:General\\_disclaimer](http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer)
- During parsing, must normalize (expand) such relative URLs

# Content seen?

---

- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles

# Filters and robots.txt

---

- Filters – regular expressions for URL's to be crawled/not
- Once a robots.txt file is fetched from a site, need not fetch it repeatedly
  - Doing so burns bandwidth, hits web server
- Cache robots.txt files

# Duplicate URL elimination

---

- For a non-continuous (one-shot) crawl, test to see if an extracted+filtered URL has already been passed to the frontier
- For a continuous crawl – see details of frontier implementation

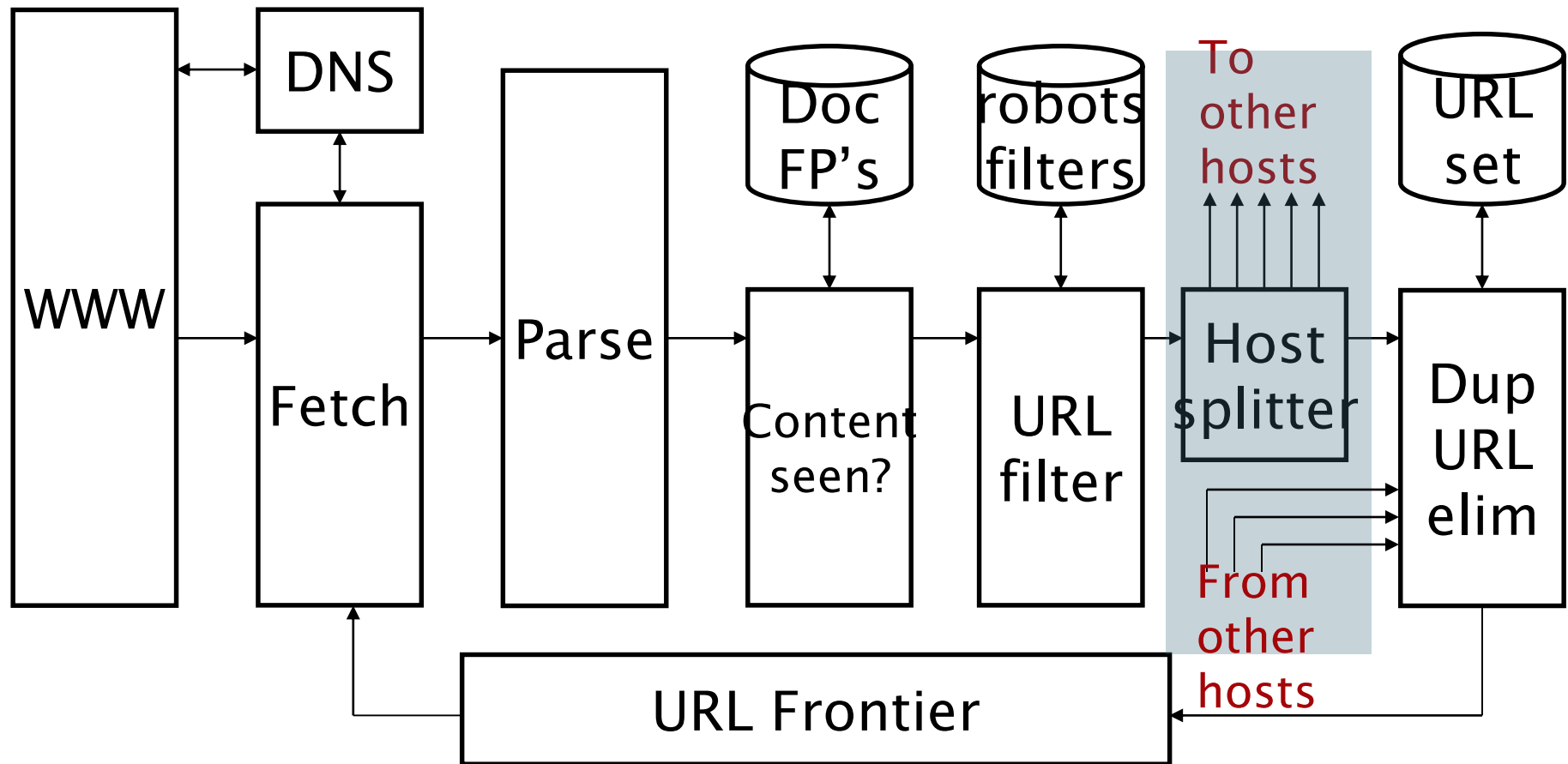
# Distributing the crawler

---

- Run multiple crawl threads, under different processes – potentially at different nodes
  - Geographically distributed nodes
- Partition hosts being crawled into nodes
  - Hash used for partition
- How do these nodes communicate?

# Communication between nodes

- The output of the URL filter at each node is sent to the Duplicate URL Eliminator at all nodes



## URL frontier: two main considerations

---

- Politeness: do not hit a web server too frequently
- Freshness: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict each other.

(E.g., simple priority queue fails – many links out of a page go to its own site, creating a burst of accesses to that site.)

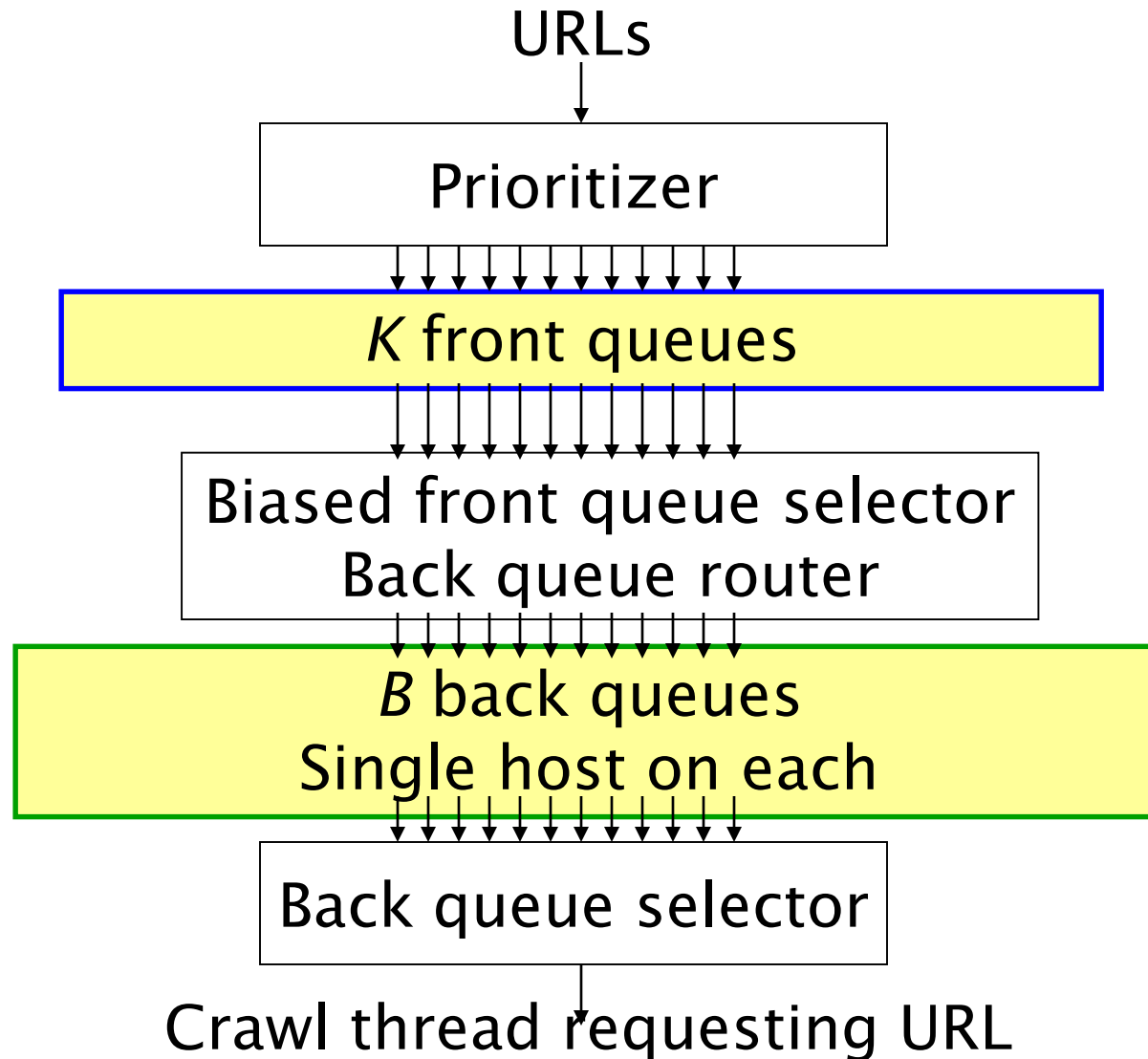
# Politeness – challenges

---

- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic: insert time gap between successive requests to a host that is  $\gg$  time for most recent fetch from that host



# URL frontier: Mercator scheme

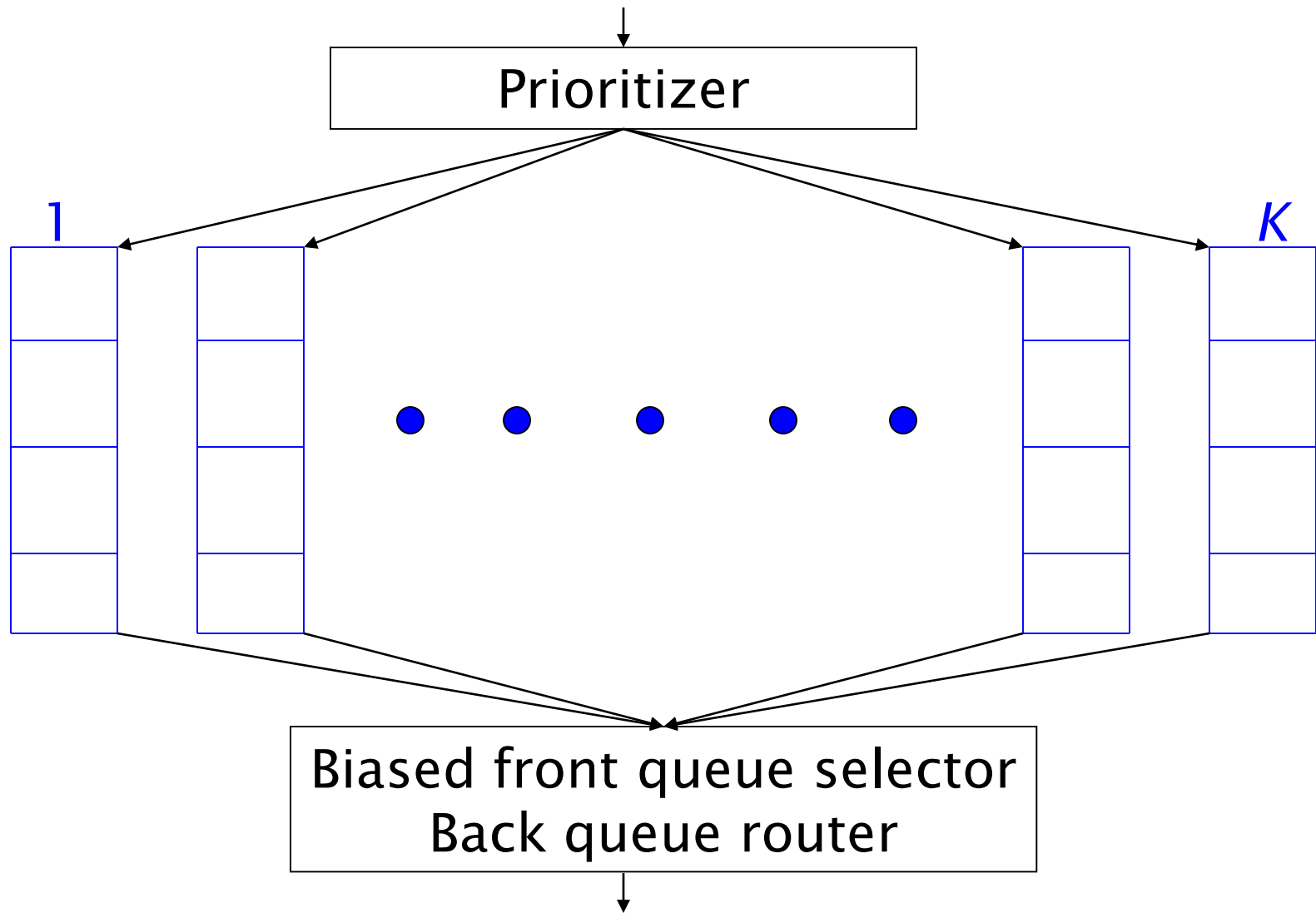


# Mercator URL frontier

---

- URLs flow in from the top into the frontier
- **Front queues** manage prioritization
- **Back queues** enforce politeness
- Each queue is FIFO

# Front queues



# Front queues

---

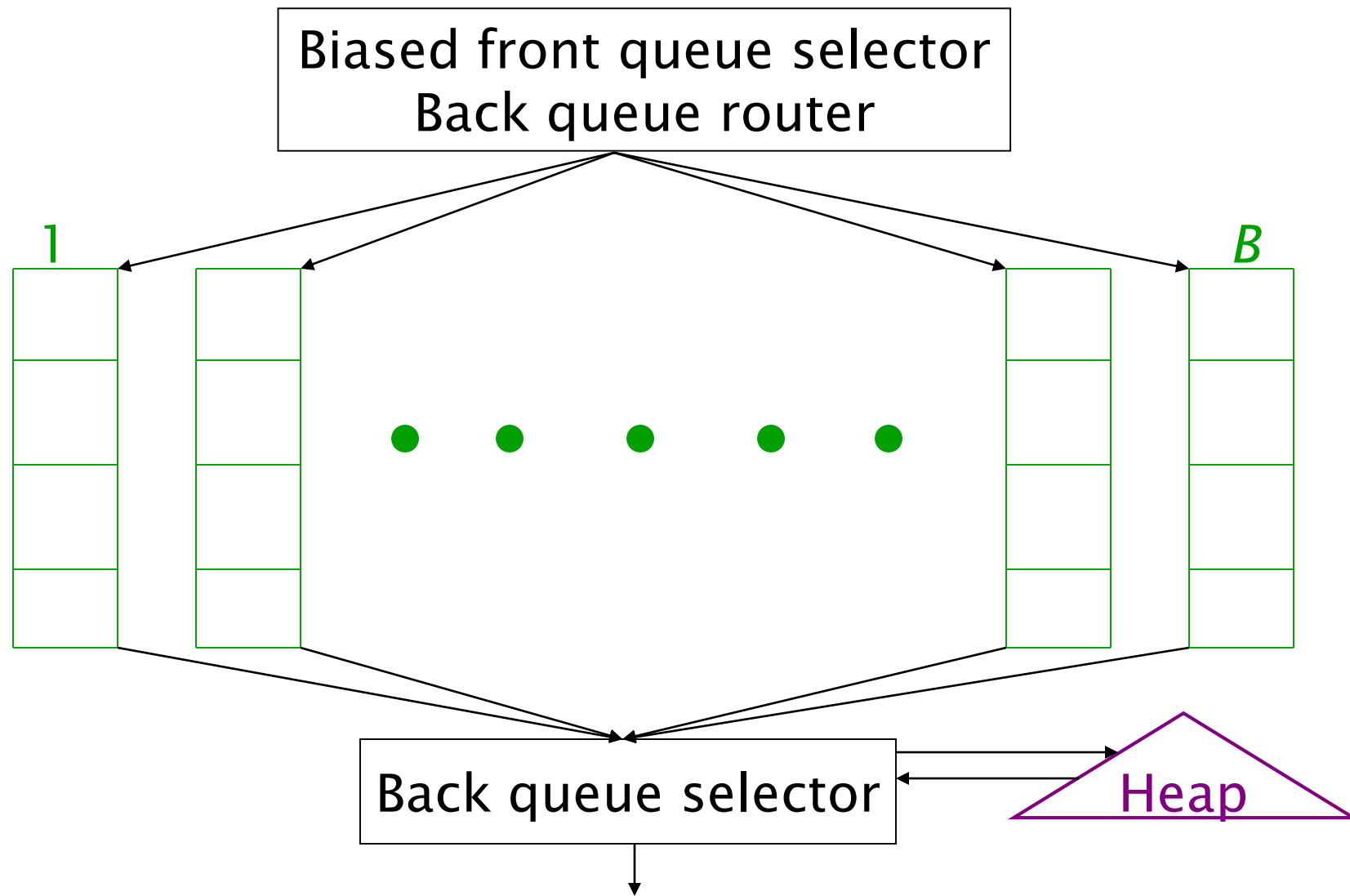
- Prioritizer assigns to URL an integer priority between 1 and  $K$ 
  - Appends URL to corresponding queue
- Heuristics for assigning priority
  - Refresh rate sampled from previous crawls
  - Application-specific (e.g., “crawl news sites more often”)

# Biased front queue selector

---

- When a back queue requests a URL (in a sequence to be described): picks a **front queue** from which to pull a URL
- This choice can be round robin biased to queues of higher priority, or some more sophisticated variant
  - Can be randomized

# Back queues



# Back queue invariants

- Each back queue is kept non-empty while the crawl is in progress
- Each back queue only contains URLs from a single host
  - Maintain a table from hosts to back queues

Host name	Back queue
...	3
	1
	<i>B</i>

# Back queue **heap**

---

- One entry for each back queue
- The entry is the earliest time  $t_e$  at which the host corresponding to the back queue can be hit again
- This earliest time is determined from
  - Last access to that host
  - Any time buffer heuristic we choose



# Back queue processing

---

- A crawler thread seeking a URL to crawl:
- Extracts the root of the heap
- Fetches URL at head of corresponding back queue  $q$  (look up from table)
- Checks if queue  $q$  is now empty – if so, pulls a URL  $v$  from front queues
  - If there's already a back queue for  $v$ 's host, append  $v$  to  $q$  and pull another URL from front queues, repeat
  - Else add  $v$  to  $q$
- When  $q$  is non-empty, create heap entry for it

# Number of back queues $B$

---

- Keep all threads busy while respecting politeness
- Mercator recommendation: three times as many back queues as crawler threads

# Connectivity servers

# Connectivity Server

---

- Support for fast queries on the web graph
  - Which URLs point to a given URL?
  - Which URLs does a given URL point to?

Stores mappings in memory from

- URL to outlinks, URL to inlinks
- Applications
  - Crawl control
  - Web graph analysis
    - Connectivity, crawl optimization
  - Link analysis

# Champion published work

---

- Boldi and Vigna
  - In the Proceedings of the WWWWeb Conference 2004
- **Webgraph** – set of algorithms and a java implementation
- Fundamental goal – maintain node adjacency lists in memory
  - For this, compressing the adjacency lists is the critical component

# Adjacency lists

---

- The set of neighbors of a node
- Assume each URL represented by an integer
- E.g., for a 4 billion page web, need 32 bits per node
- Naively, this demands 64 bits to represent each hyperlink

# Adjacency list compression

---

- Properties exploited in compression:
  - Similarity (between lists)
  - Locality (many links from a page go to “nearby” pages)
  - Use gap encodings in sorted lists
  - Distribution of gap values

# Storage

---

- Boldi/Vigna get down to an average of  $\sim 3$  bits/link
  - (URL to URL edge)
- How?



Why is this remarkable?



# Main ideas of Boldi/Vigna

---

- Consider lexicographically ordered list of all URLs, e.g.,
  - [www.stanford.edu/alchemy](http://www.stanford.edu/alchemy)
  - [www.stanford.edu/biology](http://www.stanford.edu/biology)
  - [www.stanford.edu/biology/plant](http://www.stanford.edu/biology/plant)
  - [www.stanford.edu/biology/plant/copyright](http://www.stanford.edu/biology/plant/copyright)
  - [www.stanford.edu/biology/plant/people](http://www.stanford.edu/biology/plant/people)
  - [www.stanford.edu/chemistry](http://www.stanford.edu/chemistry)

# Boldi/Vigna

- Each of these URLs has an adjacency list
- Main idea: due to templates, the adjacency list of a node is similar to one of the 7 preceding URLs in the lexicographic ordering
- Express adjacency list in terms of one of these
- E.g., consider these adjacency lists
  - 1, 2, 4, 8, 16, 32, 64
  - 1, 4, 9, 16, 25, 36, 49, 64
  - 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
  - 1, 4, 8, 16, 25, 36, 49, 64

Why 7?

Encode as (-2), remove 9, add 8

# Gap encodings

---

- Given a sorted list of integers  $x, y, z, \dots$ , represent by  $x, y-x, z-y, \dots$
- Compress each integer using a code
  - $\gamma$  code - Number of bits =  $1 + 2 \lfloor \lg x \rfloor$
  - $\delta$  code: ...
  - Information theoretic bound:  $1 + \lfloor \lg x \rfloor$  bits
  - $\zeta$  code: Works well for integers from a power law Boldi Vigna DCC 2004

# Main advantages of BV

---

- Depends only on locality in a canonical ordering
  - Lexicographic ordering works well for the web
- Adjacency queries can be answered very efficiently
  - To fetch out-neighbors, trace back the chain of prototypes
  - This chain is typically short in practice (since similarity is mostly intra-host)
  - Can also explicitly limit the length of the chain during encoding
- Easy to implement one-pass algorithm

# Resources

---

- IIR Chapter 20
- [Mercator: A scalable, extensible web crawler \(Heydon et al. 1999\)](#)
- [A standard for robot exclusion](#)
- [The WebGraph framework I: Compression techniques \(Boldi et al. 2004\)](#)