Web search engines

Paolo Ferragina

Dipartimento di Informatica

Università di Pisa

Two main difficulties

The Web:

Extracting “significant data” is difficult !!

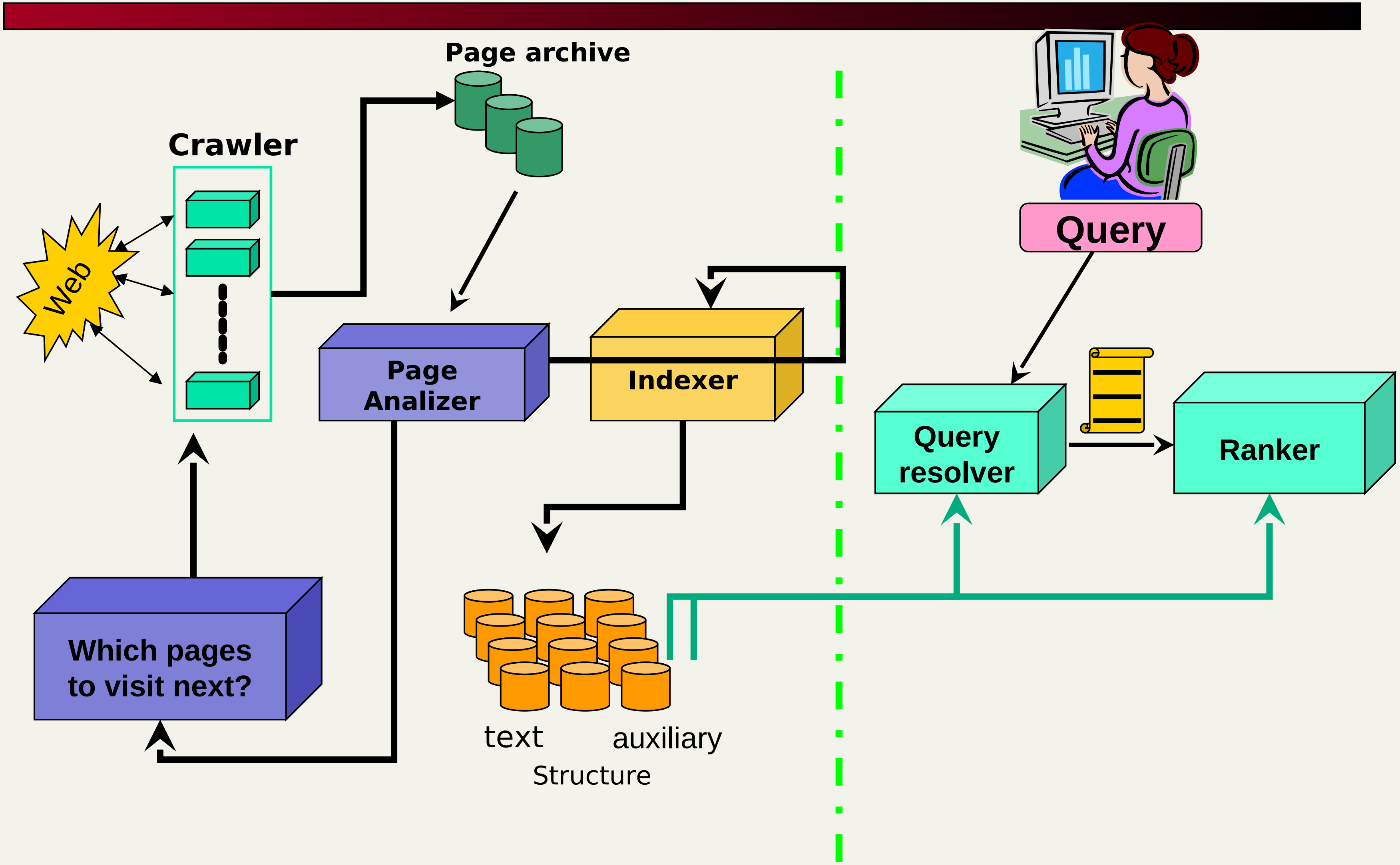
- Size: more than 1 trillion pages
- Language and encodings: hundreds…
- Distributed authorship: SPAM, format-less,…
- Dynamic: in one year 35% survive, 20% untouched

The User:

Matching “user needs” is difficult !!

- Query composition: short (2.5 terms avg) and imprecise
- Query results: 85% users look at just one result-page
- Several needs: Informational, Navigational, Transactional

The structure of a search Engine



The web graph: properties

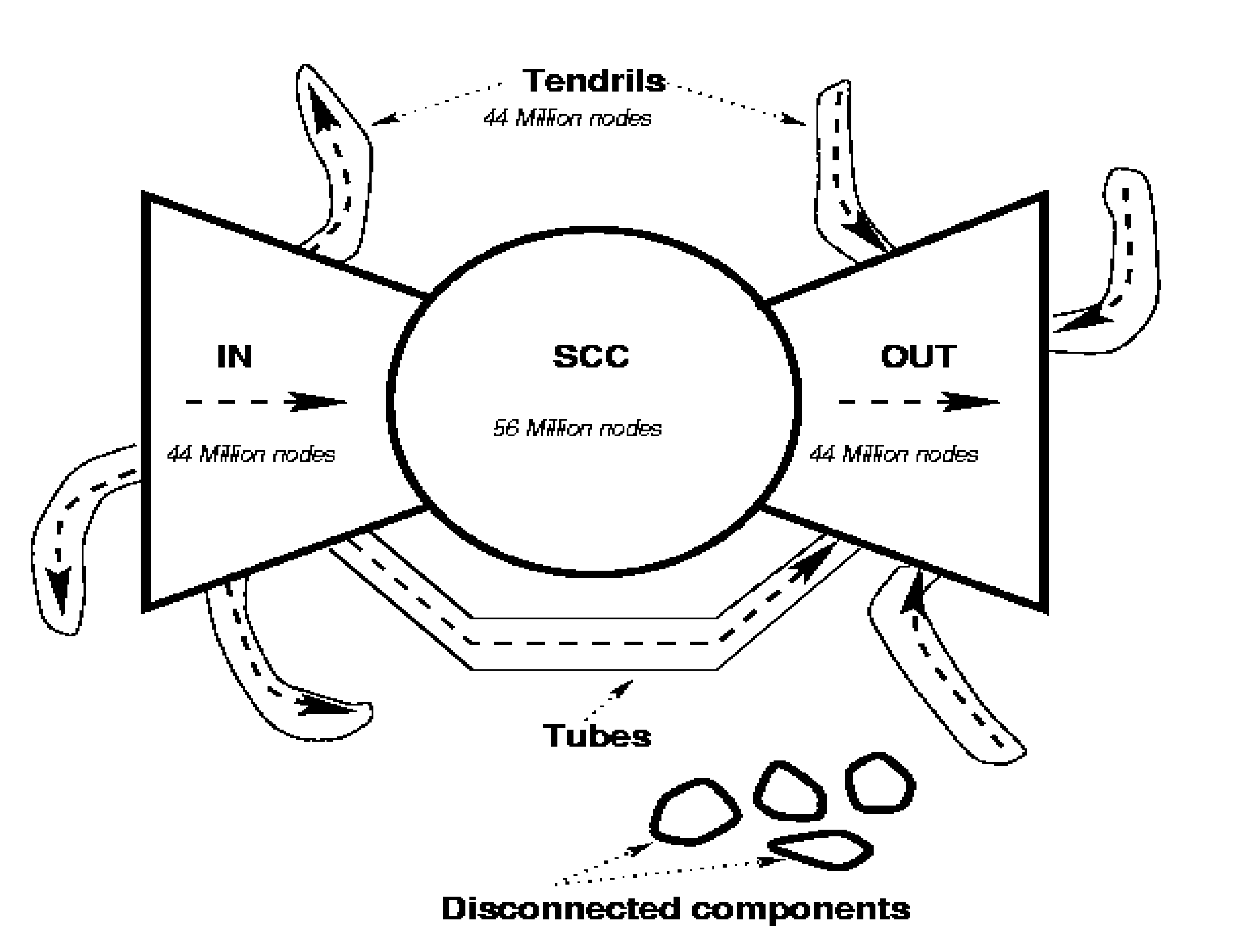
Paolo Ferragina
Dipartimento di Informatica
Università di Pisa

The Web’s Characteristics

- It’s a graph whose size is
  - 1 trillion of pages is available
    - 50 billion pages crawled (09/15)
  - 5-40K per page => terabytes & terabytes
  - Size grows every day!!
  - Actually the web is infinite… Calendars…

- It’s a dynamic graph with
  - 8% new pages, 25% new links change weekly
  - Life time of about 10 days

The Bow Tie

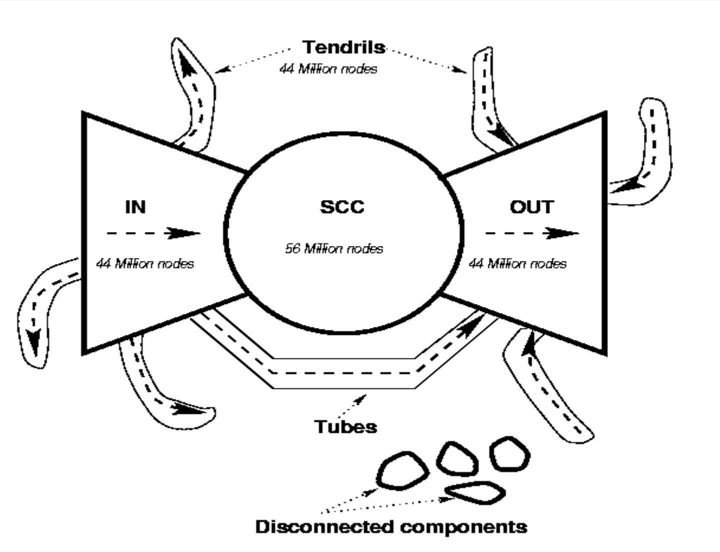

Tendrils
44 Million nodes
IN
44 Million nodes
SCC
56 Million nodes
OUT
44 Million nodes
Tubes
Disconnected components

Crawling

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa

Spidering

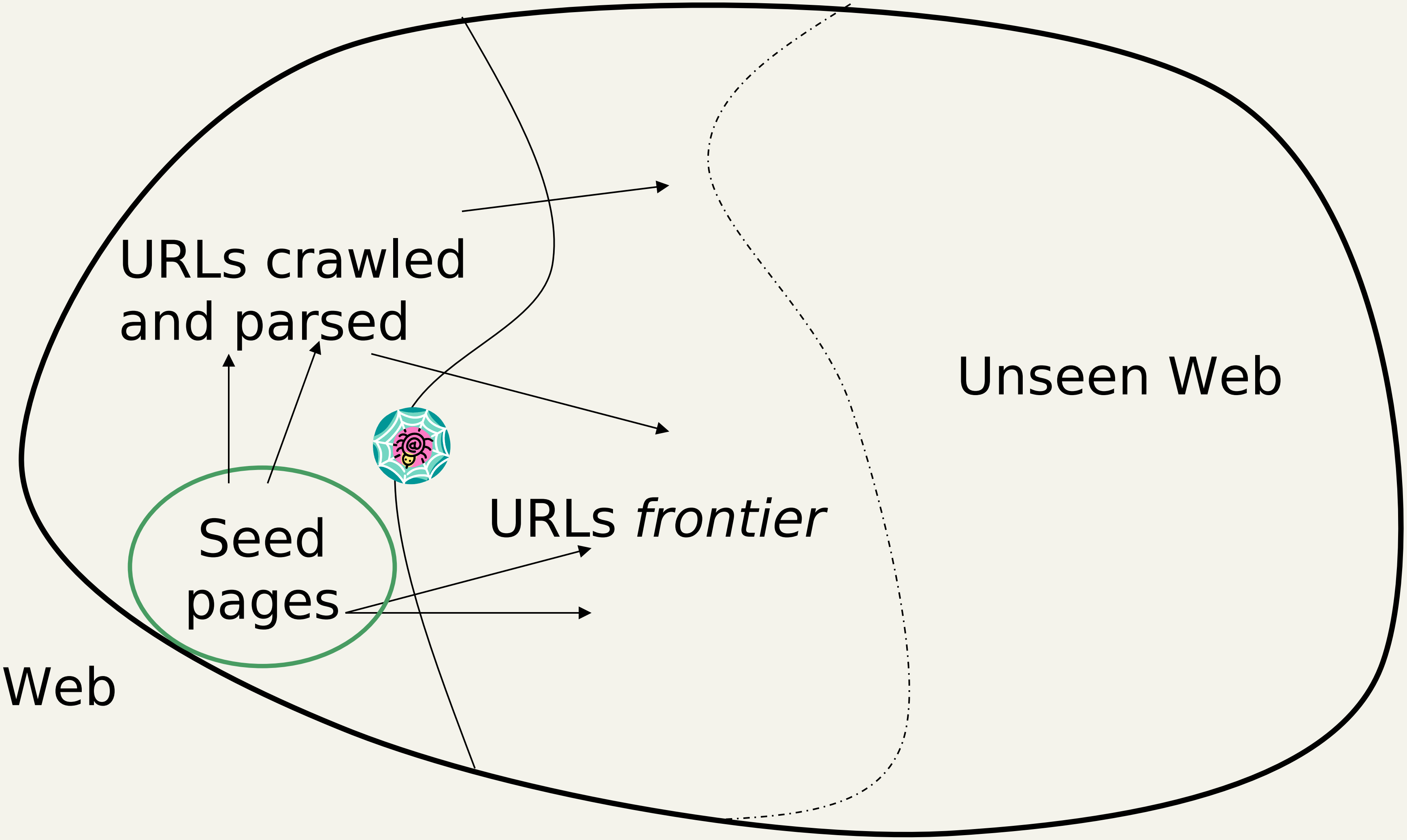
- 24h, 7days "walking" over the web graph
- Recall that:
  - Direct graph G = (N, E)
  - N changes (insert, delete) >> trillion nodes
  - E changes (insert, delete) > 10 links per node
    - Trillion entries in posting lists
    - Many more if we consider also the word positions in every document where it occurs.



Crawling Issues

- How to crawl?
  - *Quality*: “Best” pages first
  - *Efficiency*: Avoid duplication (or near duplication)
  - *Etiquette*: Robots.txt, Server load concerns (Minimize load)
  - *Malicious pages*: Spam pages, Spider traps – incl dynamically generated
- How much to crawl, and thus index?
  - *Coverage*: How big is the Web? How much do we cover?
  - *Relative Coverage:* How much do competitors have?
- How often to crawl?
  - *Freshness*: How much has changed?
  - *Frequency*: Commonly insert time gap btw host requests

Crawling picture



URLs crawled
and parsed

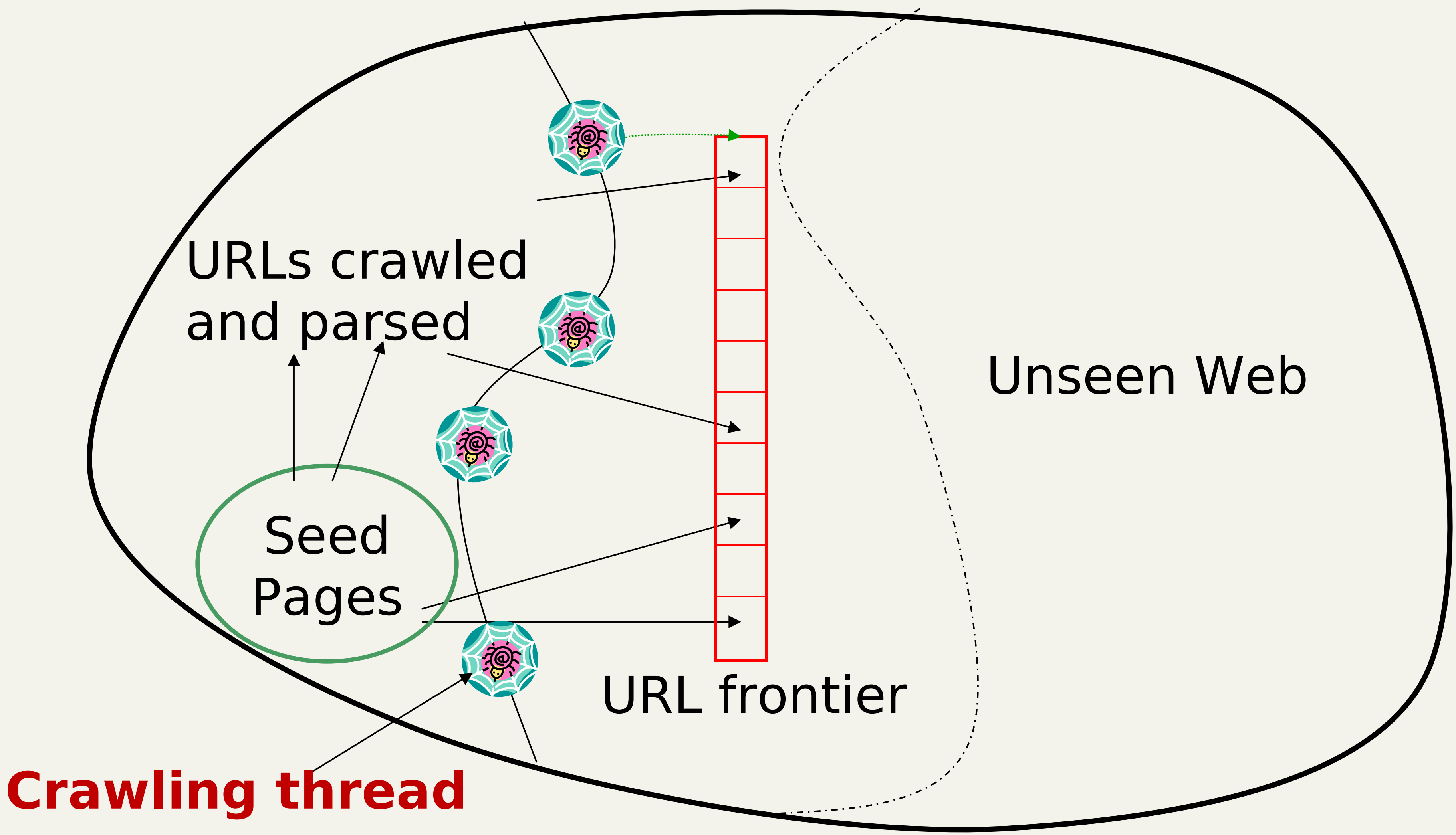
Unseen Web

Seed
pages

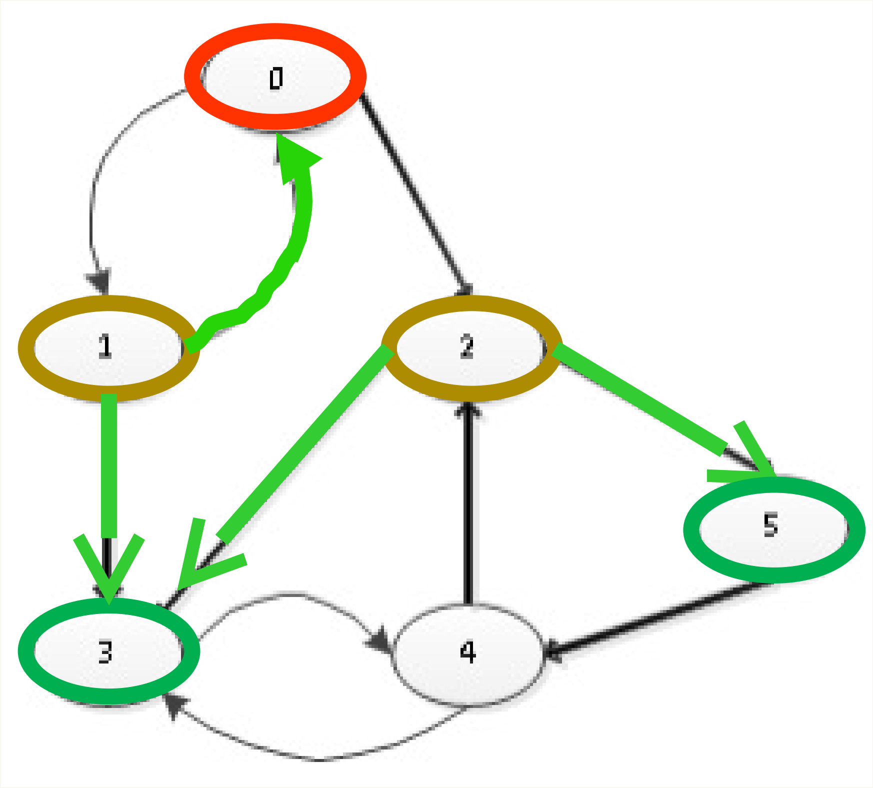
URLs *frontier*

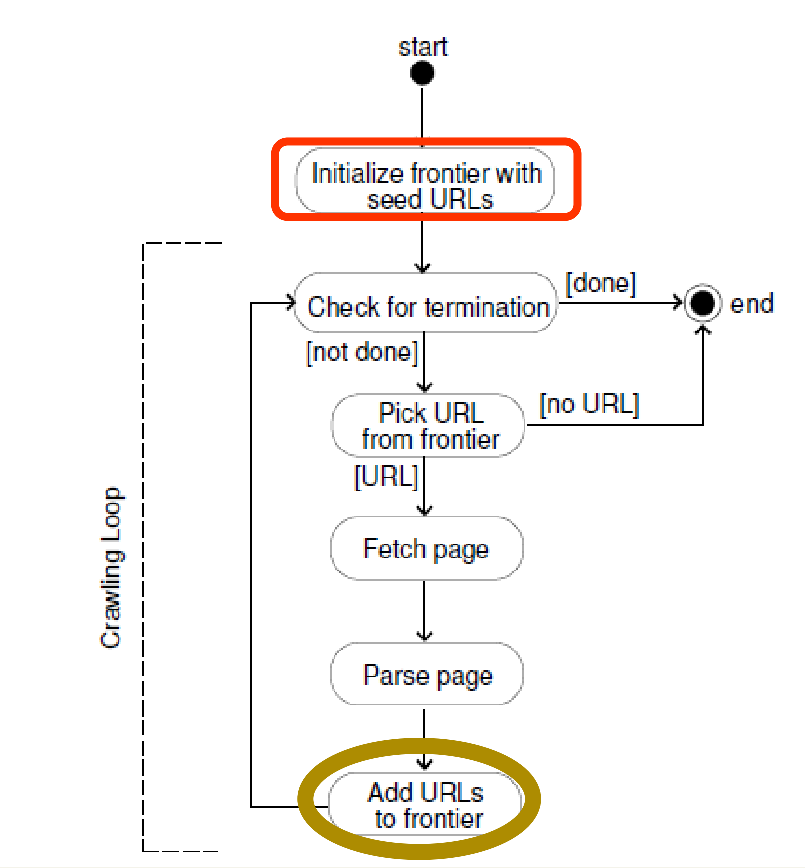
Web

Updated crawling picture

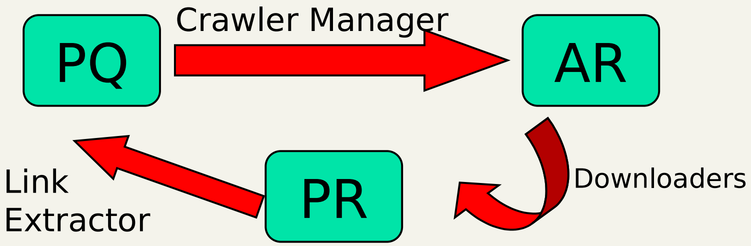


A small example





Crawler “life cycle”



One Link Extractor per page:

```
while(<Page Repository is not empty>){
   <take a page p (check if it is new)>
   <extract links contained in p within href>
   <extract links contained in javascript>
   <extract                 .....
   <insert these links into the Priority Queue>
}
```

One Downloader per page:

```
while(<Assigned Repository is not empty>){
   <extract url u>
   <download page(u)>
   <send page(u) to the Page Repository>
   <store page(u) in a proper archive,
               possibly compressed>
}
```

One single Crawler Manager:

```
while(<Priority Queue is not empty>){
   <extract some URL u having the highest priority>
   foreach u extracted {
                if ( (u ∉ “Already Seen Page” ) ||
                     ( u ∈ “Already Seen Page”  && <u’s version on the Web is more recent> )
                    ) {
                  <resolve u wrt DNS>
                  <send u to the Assigned Repository>
                  }
   }
}
```

URL frontier visiting

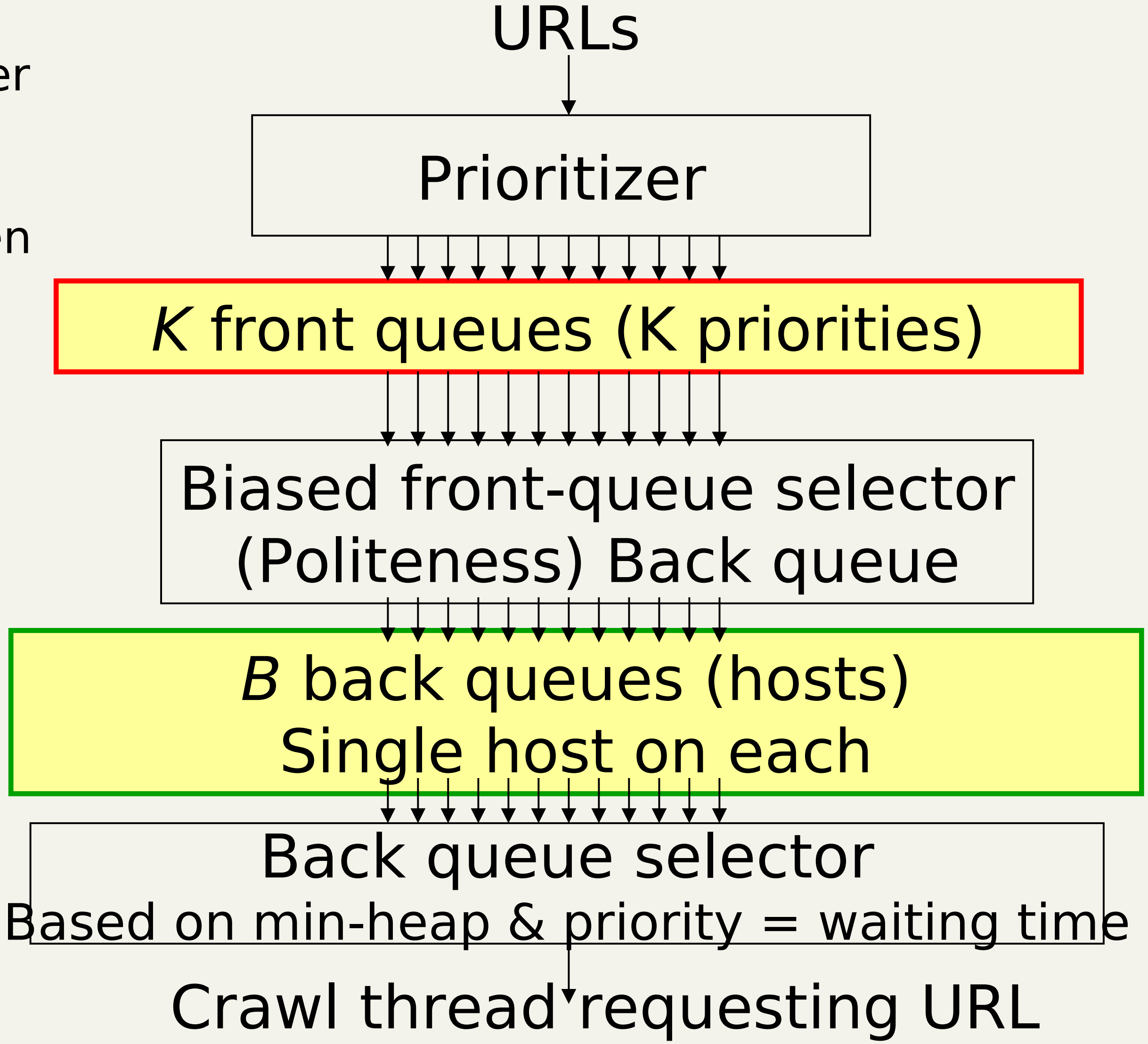
- Given a page P, define how “good” P is.

- Several *metrics* (via priority assignment):
  - BFS, DFS, Random
  - Popularity driven (*PageRank*, full vs partial)
  - Topic driven or focused crawling
  - Combined

- How to fast check whether the URL is new ?

Mercator

1. Only one connection per host is open at a time;
2. a waiting time of a few seconds occurs between successive requests to the same host;
3. high-priority pages are crawled preferentially.



URLs

Prioritizer

K front queues (K priorities)

Biased front-queue selector (Politeness) Back queue

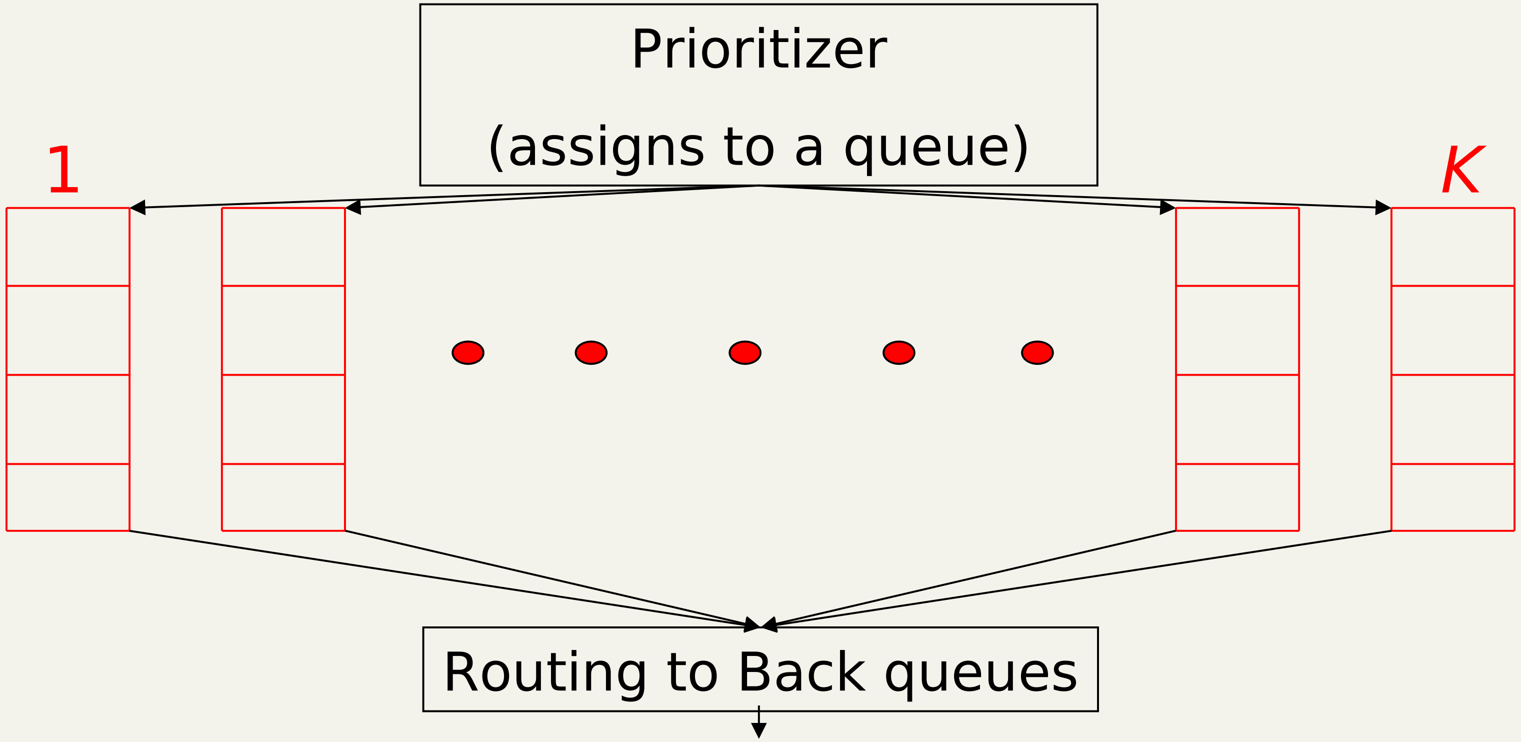
B back queues (hosts)
Single host on each

Back queue selector
Based on min-heap & priority = waiting time

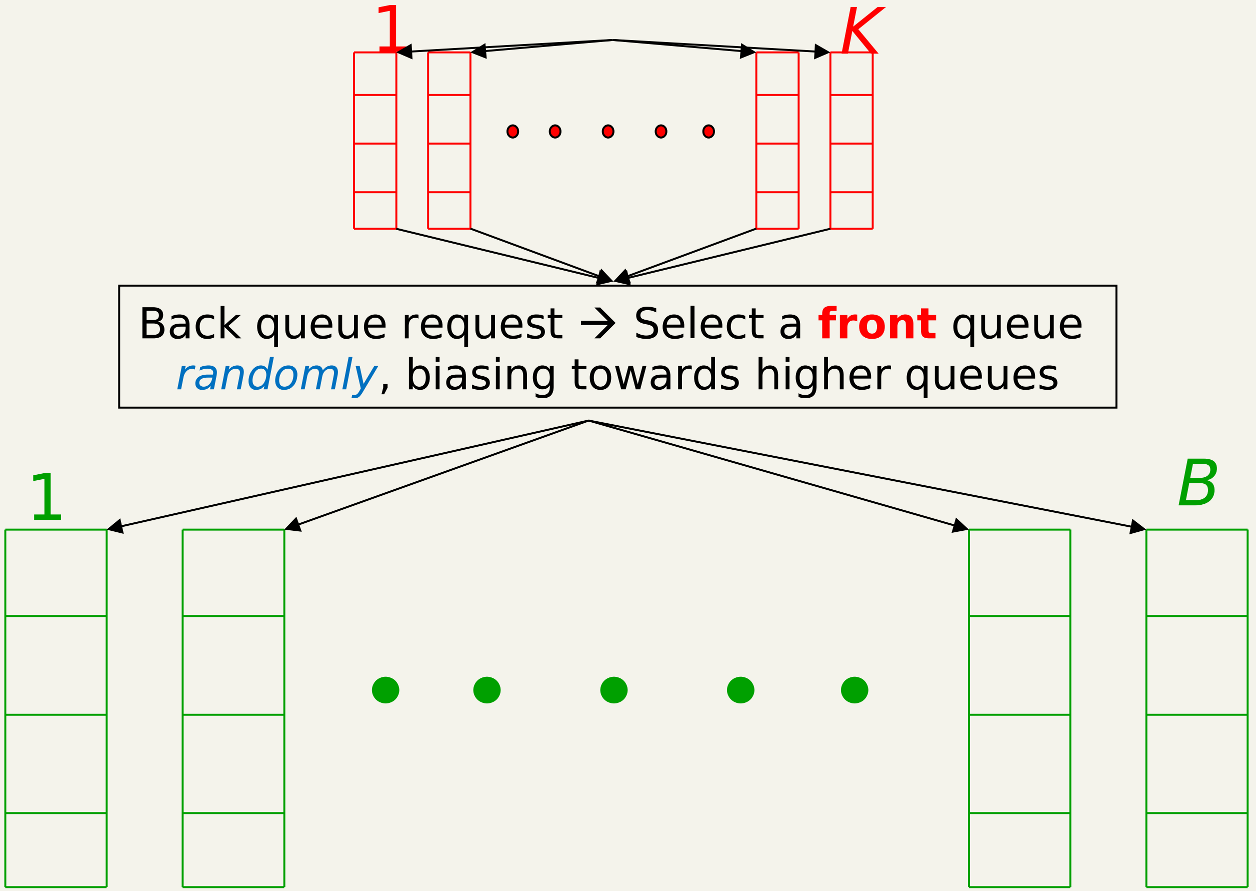
Crawl thread requesting URL

Front queues

- Front queues manage **prioritization**:
  - Prioritizer assigns to an URL an integer priority (refresh, quality, application specific) between 1 and *K*
  - Appends URL to corresponding queue, according to priority



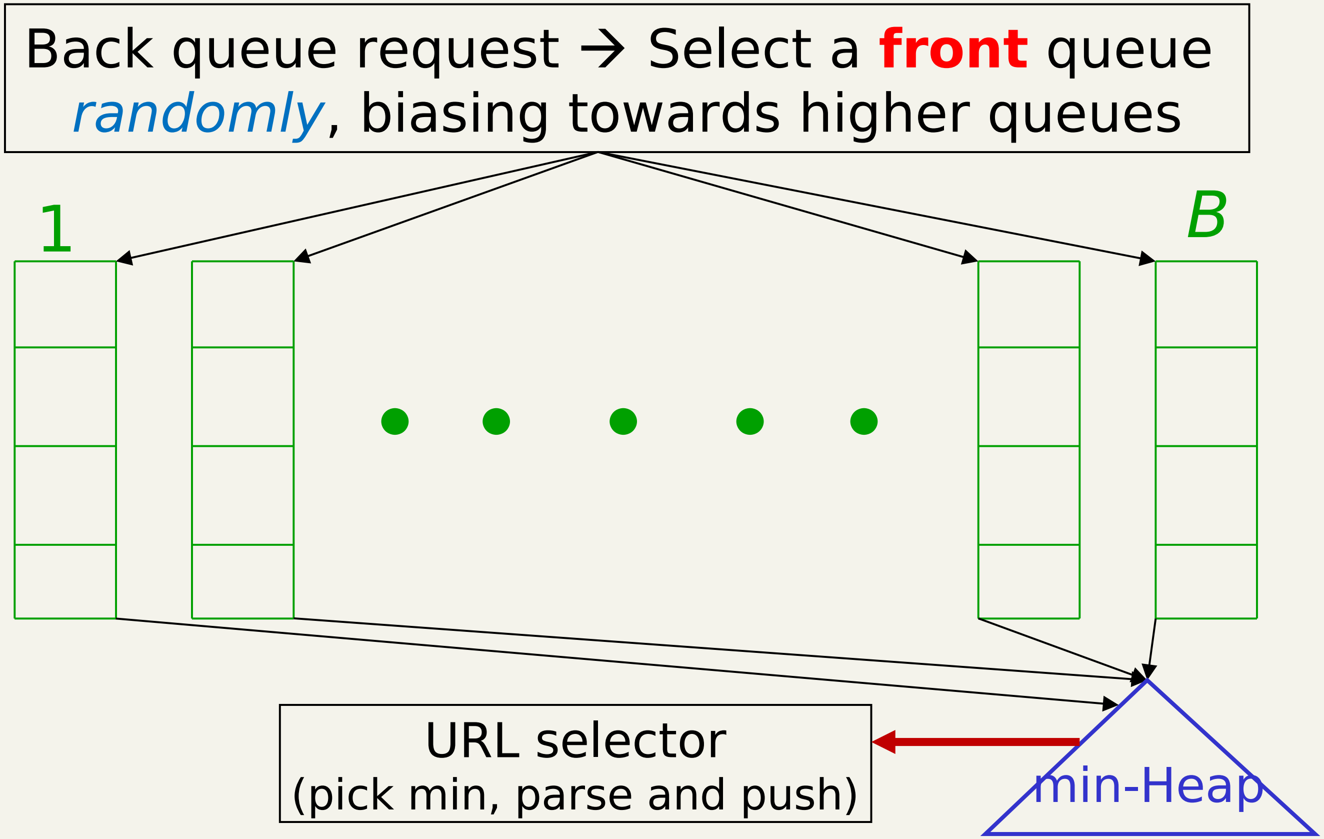
Back queues



Back queue request → Select a **front** queue *randomly*, biasing towards higher queues

Back queues

- Back queues enforce **politeness:**
  - Each back queue is kept non-empty
  - Each back queue contains only URLs from a **single** host



Back queue request → Select a **front** queue *randomly*, biasing towards higher queues

URL selector
(pick min, parse and push)

The **min-heap**

- It contains one entry per 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

The crawl thread

- A crawler seeks a URL to crawl:
  - Extracts the root of the heap: So it is an URL at the head of some back queue *q (and then removes it)*
  - Waits the indicated time t_{url}
  - Parses URL and adds its out-links to the Front queues
- If back queue *q* gets empty, pulls a **URL *v*** from some front queue *(more prob for higher queues)*
  - If there's already a back queue for *v*'s host, append *v* to it and repeat until q gets not empty;
  - Else, make *q* the back queue for *v's* host
- If back queue *q* is non-empty, pick URL and add it to the min-heap with priority = waiting time t_{url}

Keep crawl threads busy (B = 3 x threads)