Lab Manual: Module 8 – Web Crawling

Course: Information Retrieval and Web Search
Topics Covered:

- Web crawling strategies and architecture
- Politeness, robustness, and scalability
- URL normalization and deduplication
- Dynamic content handling and spam avoidance

Submission: Submit both .ipynb file and .ipynb converted to PDF

Submissions with following cases will get a zero

- Code or commented text truncated from the pdf version of the notebook
- Any compilation error in the notebook
- Missing output for any of the programming cells. There should be an output for every code cell

---

Q1. Simulate Depth-First and Breadth-First Crawling

- Load a set of 4–6 web pages
- Create a small directed graph representing a website (5–7 pages with links).
- Implement both DFS and BFS traversal.
- Print the order in which pages are visited.
- Comment on the differences in traversal behavior.

In [1]:
```
from collections import deque

## Following steps are optional to implement
    #- Load a set of 4–6 web pages
    #- Create a small directed graph representing a website (5–7 pages with links).
# Sample website graph (Output of first 2 steps)
# (You can implement first 2 steps if you want)
web_graph = {
    'Home': ['About', 'Products', 'Contact'],
    'About': ['Team'],
    'Products': ['Product1', 'Product2'],
    'Contact': [],
    'Team': [],
    'Product1': [],
    'Product2': []
}
```

In [2]:
```
# DFS
def DFS(web_graph, start_page):
    visited = set()
    stack = [start_page]
    visited_order = []

    while stack:
        cur_page = stack.pop()

        if cur_page not in visited:
            visited.add(cur_page)
            visited_order.append(cur_page)

            # add linked pages to stack
            for link in reversed(web_graph.get(cur_page, [])):
                if link not in visited:
                    stack.append(link)

    # Return the order of visited nodes
    return visited_order

# Test on given webpage
DFS(web_graph, 'Home')
```

Out[2]: ['Home', 'About', 'Team', 'Products', 'Product1', 'Product2', 'Contact']

In [3]:
```python
# BFS
def BFS(web_graph, start_page):
    visited = set()
    queue = deque([start_page])
    visited_order = []

    visited.add(start_page)

    while queue:
        # pop from front
        cur_page = queue.popleft()
        visited_order.append(cur_page)

        # add linked pages to queue
        for link in web_graph.get(cur_page, []):
            if link not in visited:
                queue.append(link)

    # Return the order of visited nodes
    return visited_order

# Test on given webpage
BFS(web_graph, 'Home')
```

Out[3]: ['Home', 'About', 'Products', 'Contact', 'Team', 'Product1', 'Product2']

Answer

The traversal behaviour in BFS and DFS are very different. If we visualize our webpages to be a tree, DFS would start at the root node, and traverse as deep as it can into its first child node before moving onto the second child node, and this pattern continues for however big the tree is.

However, in BFS this behaviour is different as we first go through every child node of the root before going further down level-by-level in the tree.

Q2. Focused Crawling with Priority Queue

- Assign a relevance score to each page (e.g., based on topic keywords).
- Use a priority queue to simulate focused crawling.
- Show how the crawler prioritizes topic-relevant pages.
- Visualize or print the crawl order.

In [4]:
```python
#Code
import heapq

##Assign a relevance score to each page (e.g., based on topic keywords).
# You can skip coding step 1 if you want and consider Sample pages with relevance scores
pages = {
    'Home': 0.2,
    'About': 0.1,
    'Products': 0.9,
    'Product1': 0.95,
    'Product2': 0.85,
    'Contact': 0.3
}

def focused_crawling(web_graph, pages, start_page):
    visited = set()

    # -score becayse heapq=min heap
    priority_queue = [(-pages.get(start_page, 0), start_page)]
    visited_order = []

    while priority_queue:
        neg_score, cur_page = heapq.heappop(priority_queue)
        score = -neg_score

        if cur_page not in visited:
            visited.add(cur_page)
            visited_order.append(cur_page)

            # Add link pages to priority queue
            for link in web_graph.get(cur_page, []):
                if link not in visited:
                    link_score = pages.get(link,0)
                    heapq.heappush(priority_queue, (-link_score, link))
```

```
    return visited_order

focused_crawling(web_graph, pages, 'Home')
```

Out[4]: `['Home', 'Products', 'Product1', 'Product2', 'Contact', 'About', 'Team']`

> Theoretical part Answer here

- The focused crawler assigns each page in the web graph a score based on relevance on how well it matches a specific topic. In our example, the product pages are more relevant than "about" or "contact". Using a max heap, we always visit the higest scored page next which is also unvisited. This means that relevant pages are crawled early, rather than just relying on the linked pages for crawling.

Q3. URL Normalization

- Write a function to normalize URLs (e.g., lowercase host, remove default ports, sort query params).
- Test it on a list of sample URLs. Add 4 more URLs to improve the list of urls.
- Explain why normalization is critical for deduplication.

In [5]:
```
#Code
from urllib.parse import urlparse, urlunparse, parse_qsl, urlencode

urls = [
    "http://ExAmPlE.com/",
    "http://example.com:80/",
    "http://example.com/a/b/../c/",
    "http://example.com/?b=2&a=1",
    "http://example.com:123/path",
    "http://Example.com/PATh?b=4&a=1",
    "http://ExAmPlE.com/a/b/",
    "https://ExAMPLE.com:443/path"
]

def url_norm(url):
    # Parse URL
    parsed_url = urlparse(url)

    # lowercase host
    s = parsed_url.scheme.lower()

    if parsed_url.hostname:
        hn = parsed_url.hostname.lower()
    else:
        hn = ""

    # remove default ports
    port = parsed_url.port
    if port and not((s=="http" and port==80) or (s=="https" and port==443)):
        hn = f"{hn}:{port}"

    path = parsed_url.path.lower()
    if path != '/' and path.endswith('/'):
        path = path.rstrip('/')

    #sort query params
    query = ''
    if parsed_url.query:
        params = sorted(parse_qsl(parsed_url.query))
        query = urlencode(params)

    unparse_url = urlunparse((s, hn, path, '', query, ''))
    return unparse_url

# loop over urls and use function
for url in urls:
    print(f"{url} -> {url_norm(url)}")
```

```
http://ExAmPlE.com/ -> http://example.com/
http://example.com:80/ -> http://example.com/
http://example.com/a/b/../c/ -> http://example.com/a/b/../c
http://example.com/?b=2&a=1 -> http://example.com/?a=1&b=2
http://example.com:123/path -> http://example.com:123/path
http://Example.com/PATh?b=4&a=1 -> http://example.com/path?a=1&b=4
http://ExAmPlE.com/a/b/ -> http://example.com/a/b
https://ExAMPLE.com:443/path -> https://example.com/path
```

> Theoretical part Answer here

- Normalization for URLs are important because we would like to reduce the number of redundant fetches of the same content. URls are normalized to a more simpler standard form, which makes it easier for duplication checking function

Q4. Duplicate Detection with Bloom Filter

- Simulate a Bloom Filter to detect duplicate URLs.
- Insert a set of normalized URLs and test for membership.
- Discuss false positives and memory efficiency.

In [6]:
```python
#Code
urls = ["http://example.com/page1", "http://example.com/page2",
        "http://example.com/page1", "http://example.com/page3",
        "http://example.com/page3", "http://example.com/page4"]
        # add more examples

import hashlib

def bloom_filter(bit_arr, item, add=False):
    size = len(bit_arr)
    num_hash = 3

    found = True
    for i in range(num_hash):
        hash_value = int(hashlib.md5(f"{item}{i}".encode()).hexdigest(),
                         16)
        idx = hash_value % size

        if bit_arr[idx] == 0:
            found=False
            if add:
                bit_arr[idx]=1
        elif add:
            bit_arr[idx]=1

    return found

bit_arr = [0]*100
# find duplicates
for url in urls:
    if bloom_filter(bit_arr, url):
        print(f"Duplicate: {url}")
    else:
        print(f"New: {url}")
        bloom_filter(bit_arr, url, add=True)
```

```
New: http://example.com/page1
New: http://example.com/page2
Duplicate: http://example.com/page1
New: http://example.com/page3
Duplicate: http://example.com/page3
New: http://example.com/page4
```

> Theoretical part Answer here

- Bloom filters can have false positives (i.e it can incorrectly report a url is a duplicate even though it hasnt been seen before), but not false negatives (detecting a already added url as new). Bloom filters are good with memory because it uses few bits per item (100 bits in our scenario) compared to storing entire url strings which would require way more memory (more than a couple of bytes), meaning bloom filters are probably at least 10x more efficient.

Part 2: Crawler Features and Robustness

Q5. robots.txt Compliance

- Write a parser that reads a sample robots.txt file.
- Determine whether a given URL is allowed for a specific User-Agent.
- Explain how this supports crawler politeness.

In [7]:
```python
#Code
# You can use the following robots_txt (if you want)
robots_txt = """
# ----------------------------------------------------------------
# 1. Global Directives (Applies to all compliant User-Agents)
# ----------------------------------------------------------------
```

```
User-agent: *
Disallow: /cgi-bin/           # Block all common binary/script directories
Disallow: /admin/             # Block primary administrative backend (MUST be secured otherwise!)
Disallow: /private/           # Block internal, private content
Disallow: /search             # Block internal search results pages (prevents duplicate content)
Disallow: /*?                  # Block URLs containing a query string (e.g., /page?id=123)
Disallow: /wp-includes/       # (If using WordPress) Block core internal files
Disallow: /feed/              # Block all common site feeds (often crawled unnecessarily)

# Disallow specific files that are large, sensitive, or contain duplicate content
Disallow: /old-sitemap.xml
Disallow: /backup-data-2025.zip
Disallow: /testing-page.html

# ---------------------------------------------------------------
# 2. Specific Directives for Google (Googlebot)
# ---------------------------------------------------------------
User-agent: Googlebot
Disallow: /api/v1/            # Block only Google from crawling specific API endpoints
Disallow: /temp/images/       # Block a temporary image folder only for Google

# Override a global block: allow Google to crawl a specific path within a disallowed directory (Optional)
# This is often unnecessary if the global rule is defined correctly, but is useful for exceptions.
# Allow: /private/public-doc.pdf

# ---------------------------------------------------------------
# 3. Specific Directives for Bing (Bingbot)
# ---------------------------------------------------------------
User-agent: Bingbot
Crawl-delay: 5                # Request Bingbot wait 5 seconds between requests (helpful for server load)
Disallow: /staging/           # Block Bing from crawling staging environments

# ---------------------------------------------------------------
# 4. Sitemap Location
# ---------------------------------------------------------------
# Always include the sitemap at the end of the file so crawlers can easily find it.
Sitemap: https://www.yourdomain.com/sitemap.xml
Sitemap: https://www.yourdomain.com/video-sitemap.xml
"""

def compliance(robots, url, user_agent):
    lines = robots.strip().split('\n')
    cur_agent=None
    disallow=[]
    allow=[]
    found_agent = False

    for line in lines:
        line = line.strip()
        if not line or line.startswith('#'):
            continue

        if ':' in line:
            key,value = line.split(':',1)
            key = key.strip().lower()
            value=value.strip()

            if '#' in value:
                value = value.split('#')[0].strip()

            if key=='user-agent':
                cur_agent=value.lower()

                if cur_agent == user_agent.lower():
                    found_agent=True
                    disallow=[]
                    allow=[]
                elif not found_agent and cur_agent=="*":
                    disallow=[]
                    allow=[]
            elif cur_agent==user_agent.lower():
                if key=="disallow":
                    disallow.append(value)
                elif key=="allow":
                    allow.append(value)

            elif not found_agent and cur_agent=="*":
                if key=="disallow":
                    disallow.append(value)
                elif key=="allow":
                    allow.append(value)

        # check allow
```

```
    for path in allow:
        if url.startswith(path):
            return True

        # disallow
    for path in disallow:
        if url.startswith(path):
            return False

    return True

# Tests
print(compliance(robots_txt,'/index.html','*'))
print(compliance(robots_txt,'/admin/','*'))
print(compliance(robots_txt,'/api/v1/ ','Googlebot'))
```

```
True
False
False
```

> Theoretical part Answer here

- This function using robots.txt supports politeness by preventing overload and respecting site owner wishes. It avoids restricted pages and follows the crawl delay. This allows for efficient use of bandwidth by avoiding indexing content the owners dont want.

Q6. Handling Malformed HTML

- Load a malformed HTML snippet.
- Use BeautifulSoup to extract text and links.
- Demonstrate resilience to broken tags and non-standard markup.

In [8]:
```
#Code
# You can prepare your html page if you want, print the html page content in the notebook to show issues
from bs4 import BeautifulSoup as bs
def malformed_html(html_tag):
    soup = bs(html_tag, 'html.parser')

    text = soup.get_text(separator=' ',strip = True)

    #extract links
    links=[]
    for a_tag in soup.find_all('a',href=True):
        links.append({
            'url':a_tag['href'],
            'text':a_tag.get_text(strip=True)
        })
    return text, links

malformed_html_tag = """
<html>
<body>
    <h1>Missing closing tag
    <p>Another unclosed paragraph
    <a href="http://example.com">Link 1</a>
    <a href='http://test.com'>Link 2</a>
    <a href=http://broken.com>No quotes</a>
    <div><p>Nested without closing
</body>
"""

malformed_html(malformed_html_tag)
```

Out[8]:
```
('Missing closing tag Another unclosed paragraph Link 1 Link 2 No quotes Nested without closing',
 [{'url': 'http://example.com', 'text': 'Link 1'},
  {'url': 'http://test.com', 'text': 'Link 2'},
  {'url': 'http://broken.com', 'text': 'No quotes'}])
```

> Theoretical part Answer here

- This parser function for an HTML tag is very efficient in extracting content from a messy html webpage. It is good enough when there are missing elemts, broken tags and odd markups in the html webpage.

Part 3: DNS, Concurrency, and Architecture

Q7. DNS Resolution and Caching

- Simulate DNS lookups with and without caching.
- Measure lookup time and show cache hits/misses.
- Discuss TTL and its impact on freshness.

In [9]:
```python
#Code
import socket
import time

dns_cache = {}

def dns_lookup(hostname, use_cache=True, ttl=300):
    cur_time = time.time()

    if use_cache and hostname in dns_cache:
        cached_ip, timestamp=dns_cache[hostname]
        if cur_time-timestamp<ttl:
            print(f"Cache Hit: {hostname}")
            return cached_ip, 0.0

    print(f"Cache miss: {hostname}")
    time_start = time.time()
    try:
        ip = socket.gethostbyname(hostname)
        elapsed = time.time()-time_start
        if use_cache:
            dns_cache[hostname]=(ip,cur_time)
        return ip,elapsed
    except:
        return None, time.time()-time_start

hostnames = ['google.com', 'github.com','google.com','github.com']

print("With Cache:")
for name in hostnames:
    ip,t = dns_lookup(name, use_cache=True)
    print(f"{name} -> {ip} ({t*1000:.0f} ms)")
print('\n')
print("Without Cache:")
for name in hostnames:
    ip,t = dns_lookup(name, use_cache=False)
    print(f"{name} -> {ip} ({t*1000:.0f} ms)")
```

```
With Cache:
Cache miss: google.com
google.com -> 192.178.192.100 (18 ms)
Cache miss: github.com
github.com -> 140.82.112.3 (11 ms)
Cache Hit: google.com
google.com -> 192.178.192.100 (0 ms)
Cache Hit: github.com
github.com -> 140.82.112.3 (0 ms)


Without Cache:
Cache miss: google.com
google.com -> 192.178.192.100 (1 ms)
Cache miss: github.com
github.com -> 140.82.112.3 (0 ms)
Cache miss: google.com
google.com -> 192.178.192.100 (1 ms)
Cache miss: github.com
github.com -> 140.82.112.3 (0 ms)
```

> Theoretical part Answer here

- a smaller TTL means there are more lookups but fresher data. A higher TTL means that there are less lookups but the data is not accurate if the IP changes more frequently.

Q8. Asynchronous Fetching

- Use asyncio or threading to simulate concurrent page fetching.
- Show how concurrency improves throughput.
- Compare with sequential fetching.

In [10]:
```python
#Code
import time
import requests
```

```python
from concurrent.futures import ThreadPoolExecutor

def fetch(url):
    response = requests.get(url)
    return response.status_code

urls = [
    'https://httpbin.org/delay/1',
    'https://httpbin.org/delay/1',
    'https://httpbin.org/delay/1',
    'https://httpbin.org/delay/1'
]

print("Concurrent")
start = time.time()
with ThreadPoolExecutor(max_workers=4) as executor:
    executor.map(fetch, urls)
con_time = time.time()-start
print(f"Time: {con_time:.2f}s")

print('\n')
start = time.time()
for url in urls:
    fetch(url)
seq_time = time.time()-start
print(f"Time: {seq_time:.2f}s")
```

```
Concurrent
Time: 1.15s


Time: 5.85s
```

> Theoretical part Answer here

- Sequential fetching proccesses 1 url at a time hence why it took 5+ seconds in total (one by one, and the time is added up). Concurrent fetching fetches all urls simeoultaneously using threading. It complestes in one second because it fetches in parallel.

Q9. Modular Crawler Architecture

- Describe the modular architecture (Scheduler, Fetcher, Parser, Storage).
- Explain the role of each module and how they interact.
- Discuss how this design supports scalability and fault tolerance.

> Answer here (Theoretical question)

- The scheduler is the main unit of the architecture. It is the decision making tool which decides which page to crawl next using the alogirthms it is provided.
- The fetcher is the tool of the architecture. it does not make the decisions, but does the heavy lifting by downloading the content of a provided URL. It uses the schedulers decision URL to fetch.
- The parser uses the fetchers output to read the content, extracting texts and other media, as well as recognizing other links in the document. The links from the parser gets fed back into the scheduler
- The storage gets the text from the parser and stores the crawled data into a database. It stores the index of the page, the connected links of different pages, and any other important meta data

Horizontal scaling using multiple machines working in parallel can be used to improve the scalability and efficiency of the architecture. This goes hand in hand with fault tolerance. If one machine fails, another machine with the same data is running, no data at any moment is lost.