Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Engineering

Name of the Student: SHASHWAT SHAH **Roll Number: O201**

SAP ID: 60004220126 **Class: C2** **Division:C2** **Batch:C22**

Subject: WEB INTELLIGENCE

DATE OF PERFORMANCE: 06/03/2025 **DATE OF SUBMISION: 06/03/2025**

EXPERIMENT NO: 04

AIM: Design a crawler to gather web information (CO2)

SOFTWARE/IDE USED: Google Colab/Jupyter Notebook

THEORY:

1. What is a web crawler and where is it used?

A web crawler (also known as a spider or bot) is a program or automated script that systematically browses the World Wide Web in order to collect data or index the content of websites. Crawlers are used primarily by search engines like Google, Bing, and others to index web pages for search results.
Uses of Web Crawlers:

- Search Engine Indexing: They gather web pages and organize them for search engines.
- Data Mining: Crawlers can extract useful data for research, analysis, or business purposes.
- Website Monitoring: Used for monitoring website content or structure for changes.
- SEO Optimization: Evaluate websites to optimize content and structure for search engine rankings.

2. Explain with a tree diagram the taxonomy of web crawlers.

```
Web Crawlers
|
├── Basic Crawlers
|   ├── Breadth-First Search (BFS)
|   └── Depth-First Search (DFS)
|
├── Focused Crawlers
|   ├── Topical Crawlers
|   └── Domain-Specific Crawlers
|
└── Universal Crawlers
    ├── Standard Crawlers
    └── Distributed Crawlers
```

3. What are basic crawlers?

Faculty In-charge: Mr. Vivian Lobo

Basic crawlers are the simplest form of web crawlers that follow predefined strategies (like BFS or DFS) to explore and collect data from websites. They do not have any special filtering mechanisms or specific targeting goals, so they crawl every page they encounter within a domain or the entire web.
Types of Basic Crawlers:

- Breadth-First Search (BFS): The crawler visits the current page and then proceeds to visit all the linked pages in breadth-first order.
- Depth-First Search (DFS): The crawler explores a page and then recursively explores all links on that page before backtracking to explore other links.

4. State several implementation issues in web crawlers.

Some common implementation issues in web crawlers include:

- Efficiency: Web crawlers need to crawl the web efficiently without overwhelming servers or consuming excessive bandwidth.
- Politeness: Crawlers must obey robots.txt guidelines and not overload websites with requests (rate limiting).
- Duplicate Content: Crawlers should identify and ignore duplicate content to avoid re-indexing the same page multiple times.
- Scalability: As the web grows, crawlers need to scale to handle millions or billions of pages.
- Handling Dynamic Content: Modern websites may have dynamic or AJAX-based content, which can be challenging for traditional crawlers.
- Handling Large Websites: Managing and crawling very large websites with millions of pages requires careful resource management and optimization.
- Data Storage: Efficiently storing crawled data (URLs, metadata, content) without consuming too much space.

5. How are universal crawlers different from preferential (i.e., focused and topical) crawlers?

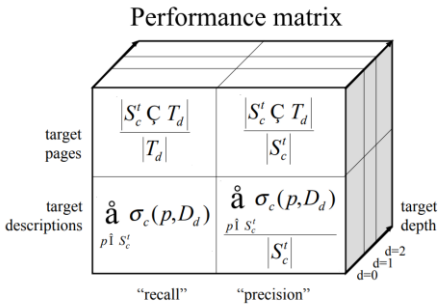
Universal Crawlers are designed to crawl the entire web with no specific focus. They aim to index as much content as possible and are typically used by search engines to gather information on a broad set of web pages.
Characteristics:

- They follow general crawling algorithms like BFS or DFS.
- They are more resource-intensive due to their broad focus.

Preferential Crawlers (Focused and Topical Crawlers) are specialized crawlers that focus on specific topics, themes, or areas of interest. They do not crawl the entire web but instead focus on a subset of pages related to a particular subject or topic.

- Focused Crawlers: These crawlers prioritize certain types of content based on relevance and filters (e.g., specific domains or topics). They are more efficient for niche applications or when a specific topic needs to be covered.
- Topical Crawlers: A type of focused crawler, topical crawlers target content related to a particular topic, such as medical, legal, or financial information.

6. Explain the 3D performance matrix used to evaluate the effectiveness of preferential crawlers, considering its three key dimensions: Target Pages (Y-axis), Target Descriptions (X-axis), and Target Depth (Z-axis).



The 3D performance matrix helps in evaluating the effectiveness of preferential (focused) crawlers by measuring their efficiency in reaching target pages, describing those pages, and navigating the depth of the target content.

Dimensions of the 3D Performance Matrix:

- Target Pages (Y-axis): This dimension measures how effectively the crawler reaches the relevant pages within the target domain or topic. It reflects the number of pages that the crawler identifies and collects which are relevant to the specified focus area.
- Target Descriptions (X-axis): This axis evaluates how well the crawler captures relevant descriptions or metadata associated with the target pages. It refers to the richness and accuracy of the content captured by the crawler (e.g., textual data, meta tags, images, etc.).
- Target Depth (Z-axis): This dimension focuses on the depth at which the crawler can explore the target pages. It considers how far the crawler can navigate into the target website or domain's structure (i.e., how many levels of sub-pages it can discover).
- Purpose: The 3D matrix helps evaluate how comprehensive, efficient, and deep the crawler is in targeting and retrieving relevant data for the specified domain or topic.

A balanced performance across all three dimensions ensures the crawler can effectively focus on the right pages, provide valuable descriptions, and explore sufficient depth to gather all relevant information.

7. Elaborate on crawler ethics and conflicts.

Crawler ethics refer to the principles and guidelines that web crawlers must follow to ensure they do not cause harm to the web, websites, or users. Web crawlers should operate responsibly to respect the rights of website owners and minimize any negative impact.

Key ethical considerations for crawlers include:

- Respect for robots.txt: Websites can define crawling rules in a robots.txt file, which specifies which pages can or cannot be crawled by automated agents. Ethical crawlers should follow these guidelines to avoid violating the website's rules.
- Politeness: Crawlers should not overwhelm a website's servers with too many requests in a short period (flooding). They must throttle their request rate and respect the crawl-delay directive in robots.txt.
- Avoiding Duplicates: Crawlers should avoid collecting duplicate data or revisiting the same page repeatedly. Efficient use of data and resources is key.

- Data Privacy and Legal Issues: Crawlers should be mindful of privacy laws such as GDPR (General Data Protection Regulation) or CCPA (California Consumer Privacy Act). They must avoid scraping personal or sensitive information.
- Fair Use of Resources: Crawlers should not place unnecessary load on servers, especially those that are already resource-constrained. Overloading servers can result in slower performance or even denial of service (DoS) attacks.
- Disclosure and Transparency: Ethical crawlers should identify themselves through proper user-agent strings and provide a contact mechanism if needed.
- Avoiding Abuse: Some crawlers may intentionally scrape and abuse content by copying it for commercial or malicious use. Ethical crawlers should not misuse the collected data.

IMPLEMENTATION:

1. Perform implementation for the same using Python programming.

```
import requests
from bs4 import BeautifulSoup
import time
import re
from urllib.robotparser import RobotFileParser
from urllib.parse import urljoin, urlparse

class WebCrawler:
    def __init__(self, seed_url, max_depth=2, delay=1):
        self.seed_url = seed_url
        self.max_depth = max_depth
        self.delay = delay
        self.visited = set()  # To keep track of visited URLs
        self.to_visit = [seed_url]  # Queue for URLs to visit

        # Ensure base URL has http/https protocol and domain
        self.base_url = urlparse(seed_url).netloc  # Extract the base
domain
        self.scheme = urlparse(seed_url).scheme  # Extract protocol
(http/https)

        # Respect robots.txt
        self.robot_parser = RobotFileParser()
        self._load_robots_txt()

    def _load_robots_txt(self):
        """Load and parse the robots.txt file for the website."""
        robots_url = f"{self.scheme}://{self.base_url}/robots.txt"
        try:
            self.robot_parser.set_url(robots_url)
            self.robot_parser.read()
        except Exception as e:
```

```
            print(f"Could not load robots.txt: {e}")

    def _is_allowed_to_crawl(self, url):
        """Check if the URL is allowed to be crawled based on
robots.txt."""
        return self.robot_parser.can_fetch("*", url)

    def _get_page_content(self, url):
        """Fetch HTML content of a page."""
        if not self._is_allowed_to_crawl(url):
            print(f"Skipping {url} (blocked by robots.txt)")
            return None

        try:
            print(f"Fetching: {url}")
            response = requests.get(url)
            if response.status_code == 200:
                return response.text
            else:
                print(f"Failed to fetch {url} with status code:
{response.status_code}")
                return None
        except requests.exceptions.RequestException as e:
            print(f"Request failed for {url}: {e}")
            return None

    def _extract_links(self, html_content, base_url):
        """Extract and normalize links from the page content."""
        soup = BeautifulSoup(html_content, "html.parser")
        links = set()
        for anchor in soup.find_all("a", href=True):
            link = anchor["href"]
            # Resolve relative URLs
            full_link = urljoin(base_url, link)
            if urlparse(full_link).netloc == self.base_url:  # Only
internal links
                links.add(full_link)
        return links

    def _extract_title(self, html_content):
        """Extract the title of the web page."""
        soup = BeautifulSoup(html_content, "html.parser")
        title_tag = soup.find("title")
        return title_tag.text if title_tag else "No Title"

    def crawl(self):
        """Main crawling function."""
```

```
        depth = 0
        while self.to_visit and depth < self.max_depth:
            current_url = self.to_visit.pop(0)
            if current_url not in self.visited:
                print(f"Visiting: {current_url}")
                self.visited.add(current_url)

                html_content = self._get_page_content(current_url)
                if html_content:
                    # Extract title
                    title = self._extract_title(html_content)
                    print(f"Page Title: {title}")

                    # Extract and add links to visit
                    links = self._extract_links(html_content,
current_url)
                    self.to_visit.extend(links)

                # Delay to avoid overloading the server
                time.sleep(self.delay)

            # Optional: Print status of visited URLs
            print(f"Visited {len(self.visited)} pages,
{len(self.to_visit)} pages left to visit.")
            depth += 1

if __name__ == "__main__":
    # seed_url = "https://tirthnisarportfolio.web.app/"  # Replace with
the starting URL
    seed_url = "https://www.djsce.ac.in/"  # Replace with the starting
URL
    crawler = WebCrawler(seed_url, max_depth=2, delay=1)
    crawler.crawl()
```

OUTPUT:

```
Visiting: https://www.djsce.ac.in/
Fetching: https://www.djsce.ac.in/
Page Title: Dwarkadas J. Sanghvi College of Engineering
Visited 1 pages, 178 pages left to visit.
Visiting: https://www.djsce.ac.in/nba
Fetching: https://www.djsce.ac.in/nba
Page Title: Dwarkadas J. Sanghvi College of Engineering
Visited 2 pages, 303 pages left to visit.
```

CONCLUSION: In this experiment, I learned the importance of ethical considerations when building and deploying web crawlers, including respecting `robots.txt` and avoiding server overload. The

performance of a crawler depends on its ability to efficiently retrieve relevant data, handle dynamic content, and overcome challenges like CAPTCHAs and access restrictions. Optimizations such as parallel crawling, intelligent request scheduling, and robust error handling are essential for improving efficiency, accuracy, and reliability in future iterations.

POST-EXPERIMENTAL EXERCISE:

1. How efficiently did the crawler perform in terms of speed, resource usage, and data retrieval?

The efficiency of a web crawler depends on several factors:

- Speed: If the crawler is able to collect data in a timely manner, this indicates a high-performance crawler. This includes how quickly it can access and download pages from the internet.
- Resource Usage: A crawler's resource consumption, including CPU, memory, and network bandwidth, is crucial. Efficient crawlers are designed to minimize system load and use bandwidth wisely.
- Data Retrieval: The crawler should be able to retrieve data without missing any relevant pages. The retrieval speed is often affected by network latency, server response time, and the crawler's ability to parallelize requests.

Improvements for efficiency can be made by using multi-threading, intelligent request scheduling, caching, and limiting the scope to avoid unnecessary retrieval.

2. Did the crawler successfully collect the intended data, and how relevant was the information gathered?

- Success in Data Collection: This refers to whether the crawler managed to fetch the required content and pages without failures. It could be impacted by issues like blocking by firewalls or CAPTCHAs.
- Relevance of Information: The gathered data should meet the crawler's objectives. For instance, if the goal is to gather product information, the crawler should specifically focus on extracting relevant fields (e.g., price, description) and avoid irrelevant content.

Relevance is key for web scraping crawlers. This can be improved through focused crawling (crawling only specific topics) and content filtering techniques to avoid irrelevant or duplicate data.

3. Were there any technical challenges, such as handling dynamic content, CAPTCHA restrictions, or blocked access?

- Dynamic Content: Some websites use JavaScript to load content dynamically (e.g., via AJAX). Traditional crawlers that only rely on HTML content may miss such dynamic data. Crawlers can be designed to handle dynamic content by simulating a browser or using headless browsers like Selenium or Puppeteer.
- CAPTCHA Restrictions: Websites use CAPTCHAs to prevent bots from accessing their pages. Overcoming CAPTCHAs involves using services like 2Captcha or building complex systems to solve them, but this can have ethical implications if done without permission.
- Blocked Access: Some websites use techniques like IP blocking, rate-limiting, or user-agent filtering to prevent crawlers. Using proxy rotation and respecting robots.txt can mitigate this problem, but it’s essential to avoid violating the website’s terms of service.

Faculty In-charge: Mr. Vivian Lobo

4. Were there any unexpected errors or failures during the crawling process, and how were they addressed?

Some common errors or failures encountered during the crawling process could be:

- 404 Errors: Pages not found due to broken links or missing content.
- Timeouts: Slow responses from web servers, causing the crawler to exceed its request timeout period.
- Server Errors (5xx): Issues like server unavailability or internal server errors might be encountered.
- Overloading the server: Sending too many requests in a short time could result in blocking or delays.

Addressing These Issues:

- Error Handling: Implement error-handling strategies, such as retries with back off (delaying retries to avoid overwhelming the server) and logging errors for analysis.
- Timeout Management: Adjust the timeout settings or handle them gracefully by retrying the request or skipping the failed link.
- Proxy Rotation: Use a pool of proxies to rotate IP addresses and prevent blocking.
- Respecting Robots.txt: Ensure the crawler adheres to web crawling rules to avoid unnecessary errors.

5. What improvements or optimizations can be made to enhance the crawler's performance, accuracy, and efficiency in future iterations?

- Parallel Crawling: Use multiple threads or distributed systems to crawl multiple pages simultaneously. This increases speed and efficiency.
- Intelligent Request Scheduling: Prioritize more relevant pages and limit the number of requests made to less important ones.
- Rate Limiting and Politeness: Implement more refined throttling to avoid overwhelming servers and respect the crawl-delay specified in robots.txt.
- Handling JavaScript: Use headless browsers (like Selenium, Puppeteer) for pages that load content dynamically via JavaScript.
- Error Recovery: Implement more robust mechanisms to detect errors and automatically retry failed requests.
- Content Parsing: Improve the accuracy of content extraction by using advanced parsing techniques like XPath or CSS Selectors to specifically target the required data.