FAST National University of
Computer & Emerging Sciences
Karachi Campus

WEB CRAWLER & SEARCH ENGINE SYSTEM

CS3001 - Computer Networks

Project Report

Submitted By:

Huzaifa Abdul Rehman (23K-0782)

Meeran uz Zaman (23K-0039)

Abdul Moiz Hossain (23K-0553)

Fall 2025

Executive Summary

This project presents a comprehensive implementation of an intelligent web crawler and search engine system, developed as part of the CS3001 Computer Networks course at FAST NUCES Karachi. The system demonstrates practical application of core networking principles including HTTP/HTTPS protocols, TCP/IP communication, DNS resolution, and ethical web crawling standards.

The crawler employs a breadth-first search (BFS) algorithm to systematically traverse web pages, respecting robots.txt directives (RFC 9309) and implementing robust error handling for network failures. Content extraction utilizes BeautifulSoup for HTML parsing, followed by natural language processing through NLTK's Porter Stemmer and stopword filtering.

The search engine implements TF-IDF (Term Frequency-Inverse Document Frequency) weighting with cosine similarity ranking, achieving efficient document retrieval through an inverted index structure. Additional features include SHA-256 content hashing for duplicate detection, thesaurus-based query expansion, and document clustering using Euclidean distance metrics.

Key Achievements:

- • Full HTTP/HTTPS protocol implementation with proper status code handling (2xx, 4xx, 5xx)
- • RFC 9309 compliant robots.txt parsing and enforcement
- • Efficient BFS crawling with FIFO queue management and O(1) duplicate detection
- • TF-IDF search engine with LTC.LTC weighting scheme and cosine similarity ranking
- • Comprehensive error handling for network timeouts, broken links, and malformed HTML

1. Project Overview

1.1 Introduction

The Web Crawler and Search Engine system is a Python-based application designed to demonstrate fundamental computer networking concepts through practical implementation. This project showcases the integration of application-layer protocols (HTTP/HTTPS), transport-layer communication (TCP/IP), and information retrieval algorithms (TF-IDF, cosine similarity) into a cohesive, functional system.

1.2 Project Objectives

The primary objectives of this project are:

- 1. Implement HTTP/HTTPS client functionality using Python's urllib library
- 2. Develop a breadth-first search web crawler with ethical crawling practices
- 3. Create a TF-IDF based search engine with relevance ranking
- 4. Demonstrate proper network error handling and resource management
- 5. Apply natural language processing techniques for content indexing

1.3 Technology Stack

Programming Language: Python 3.13

HTTP Client: urllib.request (built-in)

HTML Parser: BeautifulSoup 4 with lxml backend

NLP Library: NLTK (Natural Language Toolkit) - Porter Stemmer

Machine Learning: Scikit-Learn (Euclidean distance, clustering)
Data Processing: NumPy (matrix operations)

2. System Architecture

2.1 Architectural Overview

The system follows a modular architecture with three primary components: the Web Crawler module (WebCrawler.py), the Search Engine module (SearchEngine.py), and supporting input/output handlers. This design promotes separation of concerns and facilitates independent development and testing of each component.

2.2 Component Breakdown

HTTP Client Module

Implements TCP/IP socket communication through Python's urllib library. Handles HTTP request/response cycles with proper header management, manages connection timeouts with retry mechanisms, and processes HTTP status codes (2xx success, 3xx redirects, 4xx client errors, 5xx server errors).

URL Management System

Maintains a queue-based frontier using FIFO structure for breadth-first traversal. Tracks visited URLs using hash sets for O(1) lookup efficiency. Implements URL normalization algorithms (RFC 3986) for converting relative URLs to absolute URLs, ensuring consistent crawling behavior.

Robots.txt Parser

Fetches and parses robots.txt files according to RFC 9309 standard. Gracefully handles missing robots.txt files (proceeds with no restrictions as per protocol). Maintains allow/disallow rule sets that are checked before each HTTP request.

Content Extraction Engine

Integrates BeautifulSoup HTML parser for DOM manipulation. Traverses DOM tree to extract hyperlinks, text content, and metadata. Implements duplicate content detection via SHA-256 cryptographic hashing, ensuring each unique page is indexed only once regardless of URL variations.

Data Processing Pipeline

Performs tokenization using regex patterns to split text into individual words. Filters stopwords using predefined language-specific lists (Input/stopwords.txt). Applies Porter Stemmer algorithm for morphological analysis, reducing words to their root forms. Calculates term frequency per document and constructs document-term frequency matrix.

Search and Retrieval Module

Builds inverted index data structure for efficient term lookup. Implements TF-IDF weight calculation using logarithmic term frequency. Ranks documents using cosine similarity between query and document vectors. Supports query expansion with thesaurus for improved recall when initial results are insufficient.

3. Computer Networks Concepts Implementation

3.1 Application Layer Protocols

HTTP/HTTPS Protocol: The system implements full HTTP/1.1 protocol support through Python's urllib.request module. Each HTTP GET request follows the standard format with proper headers including Host, User-Agent, Accept-Encoding, and Connection fields. The crawler handles all standard HTTP status codes:

- • 2xx Success: Content is parsed and indexed
- • 3xx Redirects: Automatically followed by urllib
- • 4xx Client Errors: Logged as broken URLs, crawling continues
- • 5xx Server Errors: Logged as broken URLs, crawling continues

Robots.txt Protocol (RFC 9309): The crawler implements the Robots Exclusion Protocol by fetching /robots.txt from the seed domain before beginning traversal. It parses Allow and Disallow directives, building rule sets that are checked before each page fetch. Missing robots.txt files are handled gracefully with no restrictions applied, as specified by the RFC.

URL Standard (RFC 3986): URL normalization using urllib.parse.urljoin() converts relative URLs to absolute URLs, handling cases like '../page.html', '/root/page.html', and 'page.html'. A comprehensive regex pattern validates URL structure including scheme, domain, port, and path components.

3.2 Transport Layer

TCP/IP Communication: While abstracted by urllib, the system relies on TCP for reliable, connection-oriented communication. Each HTTP request triggers a TCP three-way handshake (SYN, SYN-ACK, ACK) before data transmission. TCP ensures in-order delivery and error correction through acknowledgments and retransmission.

3.3 Network Layer

DNS Resolution: Domain names are automatically resolved to IP addresses by urllib through the operating system's DNS resolver. The process involves querying recursive DNS servers, which may contact root servers, TLD servers, and authoritative name servers to obtain the target IP address.

3.4 Network Security

HTTPS/TLS Support: The system supports HTTPS URLs with automatic TLS/SSL encryption handled by urllib. Certificate validation is performed by the underlying SSL library, ensuring secure communication with HTTPS endpoints.

Content Integrity: SHA-256 cryptographic hashing generates unique 256-bit fingerprints for each page's content. This collision-resistant algorithm enables duplicate detection regardless of URL differences, ensuring index integrity.

4. Team Member Contributions

4.1 Abdul Moiz Hossain (23K-0553)

Role: Core Networking Stack & Search Algorithms

HTTP/HTTPS Implementation:

Implemented HTTP GET requests using urllib.request.urlopen() at WebCrawler.py Manages the complete request/response cycle including header generation and response parsing. Handles TCP connection establishment implicitly through urllib's socket implementation.

Breadth-First Search Crawler:

Designed and implemented the BFS crawling algorithm .Uses FIFO queue (url_frontier) for level-order traversal. Implements dequeue from front (pop(0)) and enqueue to back (append) operations to maintain BFS properties.

URL Management:

Developed URL normalization using urllib.parse.urljoin() for converting relative to absolute URLs. Implemented SHA-256 content hashing for duplicate detection. Created visited URL tracking system using hash-based dictionary for O(1) lookup.

Search Engine Algorithms:

Implemented TF-IDF calculation using formula: TF-IDF = $(1 + \log_{10}(tf)) \times \log_{10}(N/df)$. Developed cosine similarity function with L2 normalization for document ranking. Created relevance scoring system with title boosting (+0.25 for title matches).

4.2 Huzaifa Abdul Rehman (23K-0782)

Role: Content Processing Pipeline & Query Processing

HTML Parsing:

Integrated BeautifulSoup parser with lxml backend for robust HTML parsing. Handles malformed HTML through lxml's error-tolerant parser. Extracts page titles with fallback mechanism for missing title tags.

Link Extraction:

Implemented link discovery using soup.find_all('a'). Extracts href attributes and feeds them to URL normalization pipeline. Integrates with validation and scope checking before adding to frontier.

Content Extraction & Tokenization:

Developed text extraction using soup.get_text() to strip HTML tags. Created tokenization pipeline using regex to remove punctuation. Integrates with stopword filtering and word validation.

Natural Language Processing:

Integrated NLTK Porter Stemmer for word normalization. Built term-document frequency matrix for search indexing.

Query Processing:

Developed complete query processor with pipeline: tokenization → stopword removal → stemming → vectorization → cosine similarity → ranking. Implemented thesaurus-based query expansion. Integrated pickle serialization or index persistence.

Error Handling:

Implemented HTTP timeout and error handling to track broken URLs. Handles network exceptions gracefully to prevent crawler crashes.

4.3 Meeran uz Zaman (23K-0039)

Role: Protocol Compliance & Resource Management

Robots.txt Protocol (RFC 9309):

Implemented complete robots.txt parser. Fetches /robots.txt via HTTP GET, parses Allow/Disallow directives line-by-line. Handles missing robots.txt (HTTP 404) gracefully per RFC specification. Integrated enforcement check before each page fetch.

URL Validation (RFC 3986):

Created comprehensive URL validation regex supporting: http/https/ftp schemes, domain names with TLDs, localhost, IPv4 addresses, optional ports, and path/query/fragment components. Validates URLs before adding to frontier.

Domain Scope Control:

Implemented url_is_within_scope() function to prevent off-domain crawling. Extracts base domain from seed URL. Gates all extracted URLs through scope check before queuing.

Resource Management:

Created page limit enforcement system to prevent runaway crawling. Implements counter increment and loop termination when limit reached.

Data Quality Filters:

Implemented stopword filter loading from Input/stopwords.txt. Created word_is_valid() regex requiring words start with letter, end with letter/digit.

Statistics & Reporting:

Developed comprehensive statistics reporter (__str__() method) tracking: pages crawled, pages indexed, visited URLs, broken URLs, outgoing URLs, graphic URLs, and duplicate URLs. Created CSV exporter for term-document frequency matrix.

HTTP Status Code Handling:

Implemented error handling for 4xx and 5xx HTTP status codes. Tracks failed requests in broken_urls list. Ensures crawler continues despite network failures.

5. Key Implementation Details

5.1 BFS Crawling Algorithm

The crawler implements breadth-first search using a FIFO queue with the following steps: (1) Initialize frontier with seed URL, (2) Dequeue URL from front, (3) Check robots.txt compliance, (4) Send HTTP GET request, (5) Parse HTML with BeautifulSoup, (6) Extract and tokenize text content, (7) Compute SHA-256 hash for duplicate detection, (8) Extract hyperlinks, (9) Normalize and validate URLs, (10) Enqueue new URLs to back. BFS ensures important pages near the homepage are crawled first.

5.2 TF-IDF Search Algorithm

Formula: $\text{TF-IDF}(t,d) = (1 + \log_{10}(tf)) \times \log_{10}(N / df)$

TF (Term Frequency): Logarithmic scaling prevents high-frequency terms from dominating. Formula: $1 + \log_{10}(tf)$ for $tf > 0$, else 0.

IDF (Inverse Document Frequency): Reduces weight of common terms. Formula: $\log_{10}(N / df)$ where N = total documents, df = documents containing term.

5.3 Cosine Similarity Ranking

Formula: $\cos(\theta) = (Q \cdot D) / (||Q|| \times ||D||)$. **Process:** (1) Convert query and document to TF-IDF vectors, (2) L2 normalize both vectors to unit length, (3) Compute dot product, (4) Result is cosine of angle between vectors (range: 0-1). Cosine similarity is preferred over Euclidean distance because it measures directional similarity rather than magnitude, making it length-independent.

5.4 Duplicate Detection

The system uses two-level duplicate detection: **(1) URL-based:** Hash set checks if URL already visited with O(1) lookup. **(2) Content-based:** SHA-256 hashing generates unique 64-character hex fingerprint of page content. Different URLs with identical content produce same hash, enabling detection regardless of URL differences.

6. Testing and Results

6.1 Test Environments

The system was tested on three different websites: **(1) Books to Scrape** - E-commerce site with 1000+ book listings for pagination handling; **(2) Quotes to Scrape** - Quote aggregation site for short-form content indexing; **(3) Example Domain** - Simple static page for basic functionality testing.

6.2 Performance Metrics

Crawl Speed: 15-50 pages in 2-5 minutes (0.2-0.5 seconds per page). **Indexing Accuracy:** 90%+ word extraction accuracy. **Search Speed:** <1 second for 100 documents with 1000+ terms. **Memory Usage:** 50-100 MB for 50-page crawl.

6.3 Test Results: Books to Scrape

Configuration: 15-page crawl with default stopwords and thesaurus. **Results:** 15 pages crawled, 15 pages indexed (100% success), 487 unique terms, 2 duplicate pages, 0 broken URLs, 3 outgoing URLs. **Sample Query 'mystery book':** Top result = Mystery category page (0.453), Result 2 = All products page (0.321), Result 3 = Fiction category (0.287).

6.4 Error Handling Validation

The system was tested against various error conditions: HTTP 404 errors gracefully logged as broken URLs with crawler continuation, network timeouts handled via urllib's mechanism, malformed HTML successfully parsed by BeautifulSoup's lxml backend, missing robots.txt handled per RFC 9309 with no restrictions, and robots.txt Disallow directives correctly enforced.

7. Conclusion and Future Work

7.1 Project Achievements

This project successfully demonstrates comprehensive understanding and practical implementation of core computer networking principles. The web crawler and search engine system integrates HTTP/HTTPS protocols, TCP/IP communication, DNS resolution, and ethical crawling standards into a fully functional application. Key achievements include: (1) Full HTTP client with status code handling, (2) RFC 9309 compliant robots.txt parsing, (3) Efficient BFS algorithm with O(1) duplicate detection, (4) Production-grade TF-IDF search engine with cosine similarity, (5) Robust error handling for network failures, and (6) Natural language processing pipeline with stopword filtering and Porter stemming.

7.2 Learning Outcomes

Through this project, the team gained hands-on experience with application-layer protocol implementation (HTTP/HTTPS, robots.txt, URL parsing), transport-layer concepts (TCP reliability, connection management), network error handling and fault tolerance strategies, information retrieval algorithms (TF-IDF, cosine similarity, inverted indices), natural language processing techniques (tokenization, stemming, stopword removal), and software engineering best practices (modularity, error handling, documentation).

7.3 Future Enhancements

Future enhancements include: multi-threading for parallel crawling (10-100x speed improvement), JavaScript rendering integration (Selenium/Playwright), distributed architecture using message queues (RabbitMQ/Kafka), database integration (PostgreSQL/Elasticsearch) for persistent indices, advanced NLP with word embeddings (Word2Vec, BERT), PageRank algorithm for link-based ranking, web interface (Flask/Django) for easier interaction, and incremental crawling with change detection.

7.4 Final Remarks

This project successfully bridges theoretical networking concepts from CS3001 with practical software development. The web crawler and search engine demonstrate that complex distributed systems like Google, Bing, and DuckDuckGo are built on foundational principles of HTTP, TCP/IP, DNS, and information retrieval algorithms. The modular architecture, comprehensive error handling, and adherence to RFC standards showcase professional software engineering practices. The system is production-ready for small to medium-scale crawling tasks and serves as a solid foundation for more advanced features. Through collaborative development, the team successfully integrated networking (Abdul Moiz), content processing (Huzaifa), and protocol compliance (Meeran) into a cohesive, functional system that meets all project objectives.