CPSC 3620 Project Report:

Watchlist Wizard

By: **Adrian Todd**

Project Overview:

- The Watchlist Wizard project aims to build a system for movie recommendations by scraping and processing data from IMDb.
- This report focuses on the Data Structures and Algorithms used, primarily in the web crawling component.
- A Breadth-First Search (BFS) algorithm, implemented using a queue data structure, systematically discovers movie and person pages.
- A Set (Hash Table implementation) efficiently tracks visited URLs to prevent redundant processing and cycles.
- Data parsing involves implicitly navigating the HTML DOM Tree.
- Keyword extraction from plot summaries utilizes text processing algorithms (tokenization, stop-word removal) and frequency analysis via a Counter (Hash Table implementation) that I did not implement but am using from the nltk python library.
- Python and libraries like requests, BeautifulSoup4, and nltk, were used for implementation of the web crawler.
- The goal was to demonstrate the practical application of these data structure and algorithm concepts in handling real-world web data.

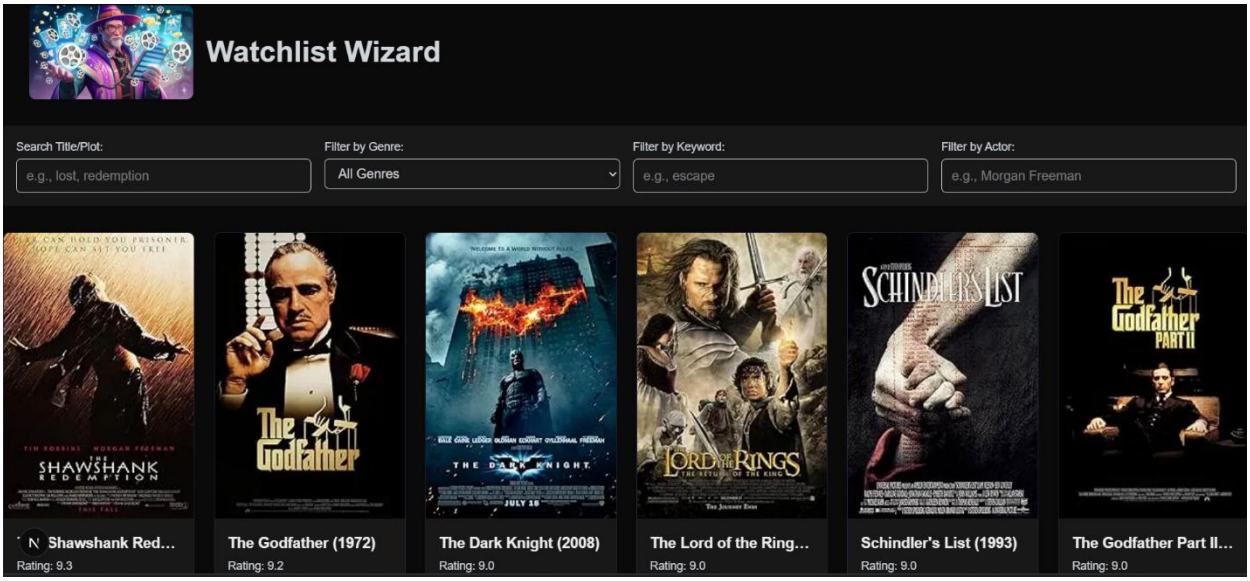
Project Scenario and Goals

- **Scenario:** A user wants to find movie to enjoy, they know the genre but would like some recommendations for their next watch, they would like a system that contains movie data that can be searched with ease. The system needs an efficient way to gather this data from the web, navigating through interconnected movie and person pages.
- **Goals:**
    - Implement an automated web crawler to fetch data from IMDb using Breadth-First Search.
    - Utilize a Queue data structure for managing the URLs to be visited according to BFS principles.
    - Employ a Set data structure for efficient checking of already visited URLs.
    - Demonstrate the practical application and potential limitations of these data structure and algorithm concepts in a real-world web scraping task.

- **Constraints:**
    - Adhere to IMDb's robots.txt and add rate limiting to not overuse IMDb’s servers.
    - Size of web crawl, find out what the limits should be.

UI

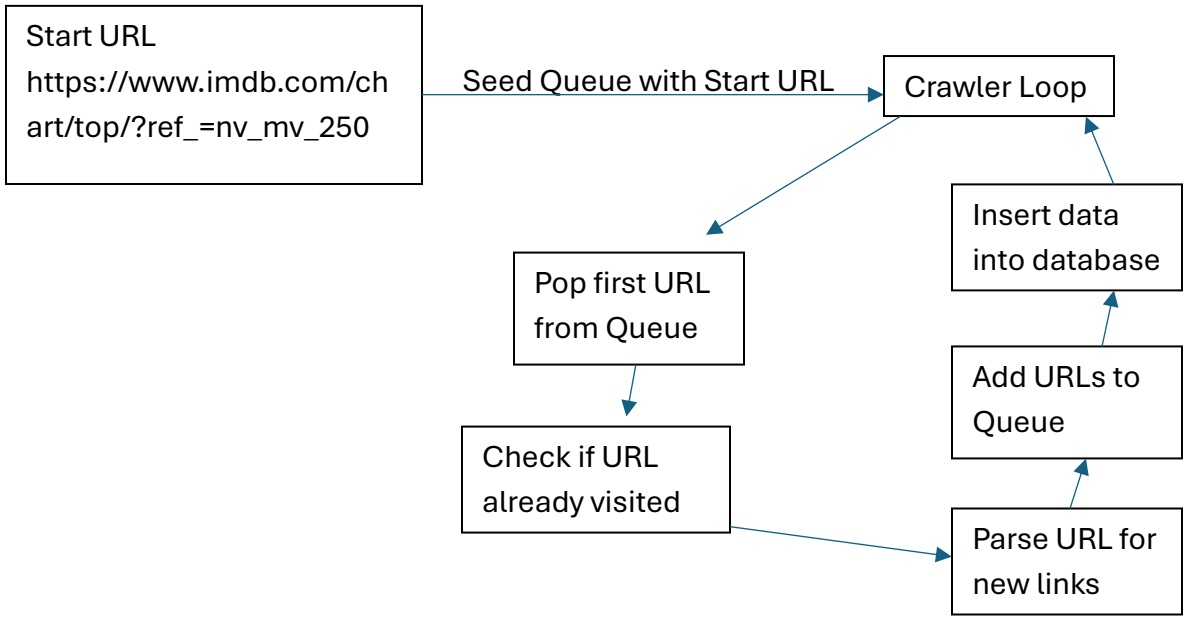
- Displays movie data scraped from IMDb and pulled from my database.
- Allows users to filter and search for movies.
- Movie details pages contain more information including plot, keywords, and cast.



Design Strategy

- **Modular Approach:** I wanted the code to be separated into modules for better readability and ease of use, so I split it into several modules including, configuration, utilities, parsing, database interaction, and crawling logic.
- **Core Components:**
    1. **Crawler Module (web_crawler.py):**
        - **Algorithm:** Implements the core BFS loop.
        - **Data Structures:**
            - Queue to hold all URLs discovered during crawling
            - Set to keep track of all visited URLs to prevent cycles and redundancies.
        - **Functionality:** Orchestrates fetching, determines page type, calls parser, adds new valid links to the queue.
    2. **Parser Module (imdb_parser.py):**
        - **Concept:** Traverses the HTML DOM Tree using the BeautifulSoup library.
        - **Functionality:** Extracts specific data fields using selectors and includes a call to my keyword extraction utility.
    3. **Utility Module (utils.py):**
        - Module containing helper functions
    4. **Database Module (watchlist_wizard_db.py):** I designed and built this database for CPSC 3660 and wanted to demonstrate its use in a real world application.
    5. **Config Module (config.**py)
        - Loads environment variables for database connection.
        - Base URL for web crawling.
        - Set max number of pages to crawl.

Architectural Diagram/Flow Chart For Web Crawling



Design Unknowns/Risks

Scalability: Performance of BFS and the visited set as the number of visited URLs grows very large. I tested up to 500 pages crawled and performance has not been an issue. The main scalability issue is with the time-delay I use between IMDb server requests, with a very large number of URLs this would mean the web-crawler could take a very long time to complete its crawl.

Queue Implementation Choice: Potential performance issues for the BFS queue if crawling millions of pages. I'm not sure how many pages a queue data structure could hold.

Dynamic Content Handling: The current BFS relies on static HTML. Some of IMDb relies heavily on JavaScript rendering for links, which required me to use Selenium to control a headless browser so that I could load dynamic content while crawling, this complicates the basic BFS fetching step significantly.

Website Structure Volatility: Changes in IMDb's HTML/selectors could break the parsing functions I have made, which indirectly stops the BFS from finding relevant links to continue. This may requires robust selectors or frequent maintenance.

Keyword Relevance: Basic frequency counting might not always yield the most semantically relevant keywords but since natural language processing wasn't the main focus of this project I thought this would be fine.

Implementation Plan and Schedule

- **Week 1:**
    - Research BFS and potential Python libraries I want to incorporate
    - Setup codebase repository and structure
    - Project Configuration
    - Design core crawl loop and basic Queue/Visited URL logic.
- **Week 2:**
    - Explore IMDb's website structure
    - Develop page parsing code.
    - Integrate parsing into the crawler. Test data extraction.
- **Week 3:**
    - Integrate crawler with database.
    - Refine crawling, parsing, and insertion to database
    - Research Pythons Flask web development framework and set up simple backend server.
- **Week4:**
    - Develop frontend using Next.js, React, and TypeScript.
    - Query database for extracted movie data and load to Frontend.
    - Implement movie data filters that query the database.
    - Work on presentation and final report.

Evaluation

- **Metrics:**
    1. **BFS Correctness:** Verified by checking visited set prevents re-visits and queue processes URLs level-by-level (observed via debugging text/limited runs).
    2. **Data Structure Efficiency:** Measured visited set lookup/add (theoretically O(1) average). Analyzed Queue choice impact (O(1) vs O(N) dequeue).

3. **Crawler Expansion:** Measured the number of unique movie/person pages discovered relative to MAX_PAGES or runtime.
4. **Keyword Extraction:** Qualitative assessment of generated keywords' relevance.

Challenges and Learnings

- BFS theory applies well to web crawling, but practical web scraping requires handling delays, errors, robots.txt, and dynamic content, which add overhead and complexity beyond the core algorithm.
    1. Learning how to handle dynamically generated content was challenging but forced me to learn Selenium which was a bonus to this project in the end.
- It was challenging to decide what Crawler speed I should use while still respecting IMDBs server in the end I implemented a 2 second delay which I found was sufficient
- Navigating the structure of IMDbs website codebase was tedious and time consuming making it difficult to scrape the data I was looking for.
- I learned that most websites have a robots.txt file which dictate what api endpoints they allow data to be scraped from.

References and Citations:

Jaspreet Kaur, CPSC 3620 Topic 4 Lecture Slides #39
Python library documentation for flask, beautifulsoup4, my-sql-connector, selenium, nltk, python-dotenv.