Spider Scout

WebCrawler for searching and scouting valuable info across the web.

Crawler Components:

- URL Frontier Module
- Downloader Module
- Parser Module
- Indexer Module
- Scheduler Module
- Logger Config
- Robots.txt Handler

URL Frontier

The frontier.py file in the spider_scout project implements the URLFrontier class, which manages the queue of URLs to be crawled. It uses a priority queue to handle URLs based on their priority and depth, tracks visited URLs to avoid duplicates, and provides thread-safe operations for adding and retrieving URLs.

```python
class URLFrontier:
    def __init__(self):
        self.lock = threading.Lock()
        self.logger = logging.getLogger(__name__)
        self.logger.debug("Initialized URLFrontier")

    def display(self):
        print("VISITED:\n", self.visited)
        print("FRONTIER:")
        with self.lock:
            frontier_list = list(self.frontier.queue)
            for item in frontier_list:
                print(item)
    # def add_url(self, url, priority=0):
    #     if url not in self.visited:
    #         self.frontier.put((priority, url))
    #         self.visited.add(url)
    #         self.logger.info(f"Added URL to frontier: {url}")

    def add_url(self, url, depth, priority=0):
        try:
            parsed_url = urlparse(url)
            if parsed_url.scheme and parsed_url.netloc:
                normalized_url = url.rstrip('/')
                with self.lock:
                    if normalized_url not in self.visited:

                        # FOR BFS TRAVERSAL
                        ## When adding URLs to the frontier
                        # priority = depth  # Lower depth gets higher priority (processed earlier)
                        # self.frontier.put((priority, depth, url))

                        self.frontier.put((priority, depth, normalized_url))
                        self.visited.add(normalized_url)
                        # self.logger.info(f"Added URL: {normalized_url}")
        except Exception as e:
            self.logger.error(f"Error adding URL to frontier: {str(e)}")

    def is_empty(self):
        with self.lock:
            return self.frontier.empty()

    def has_next(self):
        with self.lock:
            return not self.frontier.empty()
```

Scheduler Module

The scheduler.py file in the spider_scout project implements the Scheduler class, which coordinates the web crawling process. It manages the allocation of URLs to downloader and parser threads, ensures adherence to robots.txt rules, and tracks crawling progress. The scheduler continuously assigns URLs from the URL frontier to available downloaders and parsers until the crawl reaches the specified depth or all URLs are processed.

```python
class Scheduler:

    def crawl(self, seed_url, max_depth, respect_robots_txt=True):
        try:
            # Initialize crawling
            self.logger.info("Starting crawl...")
            self.url_frontier.add_url(seed_url, depth=0)
            if respect_robots_txt:
                self.robots_txt_handler.fetch_robots_txt(seed_url)

            # Start worker threads
            for worker in self.downloaders + self.parsers:
                worker.start()

            # total_urls = 1  # Start with the seed URL
            # processed_urls = 0

            while True:
                # # Check if max depth reached
                # if processed_urls >= max_depth:
                #     break

                # Try to get next URL from frontier
                url, depth = self.url_frontier.get_next_url()
                if url is not None and depth is not None:
                    if depth > max_depth:
                        continue  # Skip URLs beyond max depth
                    if (not respect_robots_txt) or (self.robots_txt_handler.is_allowed(url)):
                        self.downloader_queue.put((url, depth))  # Pass depth to downloader
                    if self.progress_callback:
                        self.progress_callback(depth, max_depth)
                else:
                    # No URLs left to process
                    if self.downloader_queue.empty() and self.parsers_queue.empty():
                        all_idle = all(downloader.state == 'idle' for downloader in self.downloaders)
                                   all(parser.state == 'idle' for parser in self.parsers)
                        if all_idle:
                            break
                    else:
                        time.sleep(1)

            # Signal termination
            for _ in self.downloaders:
                self.downloader_queue.put(None)
            for _ in self.parsers:
                self.parsers_queue.put(None)
```

Downloader Module

The downloader.py file in the spider_scout project defines the Downloader class, which is responsible for downloading web pages. The class uses multithreading to handle multiple download tasks concurrently. It fetches the HTML content of URLs from the scheduler's queue, and if successful, forwards the content to the parser queue. The downloader retries fetching URLs up to a specified number of attempts, employing exponential backoff in case of failures.

```python
class Downloader(threading.Thread):
    def run(self):
                task = self.scheduler.downloader_queue.get(timeout=1)  # Waiting for a task from the (
                if task is None:
                    #
                    self.scheduler.downloader_queue.task_done()
                    #
                    break

                self.state = 'running'
                url, depth = task
                html_content = self.fetch(url)
                if html_content:
                    self.scheduler.parsers_queue.put((html_content, url, depth))  # Forwarding the htr
                self.state = 'idle'

                self.scheduler.downloader_queue.task_done()  # Mark the task as done
            except queue.Empty:
                continue
            except Exception as e:
                self.logger.error(f"Error in downloader run loop: {str(e)}")

    def fetch(self, url, max_attempts=3):
        """
        Fetches the URL and returns the HTML content if successful.
        Retries up to `max_attempts` in case of failure.
        """
        headers = {'User-Agent': self.user_agent}
        for attempt in range(max_attempts):
            try:
                self.logger.info(f"Attempt {attempt + 1} Fetching: {url}")
                response = requests.get(url, headers=headers, timeout=10)  # Using a 10 second timeout
                response.raise_for_status()  # Raise exception for HTTP errors (4xx, 5xx)

                self.logger.info(f"Successfully fetched: {url}")
                return response.text  # Return the HTML content of the page

            except requests.RequestException as err:
                self.logger.error(f"Error fetching {url}: {str(err)}")
                if attempt < max_attempts - 1:
                    # Implementing exponential backoff on retries
                    time.sleep(self.delay * (2 ** attempt))
                else:
                    self.logger.error(f"Failed to fetch {url} after {max_attempts} attempts")
                    return None  # Return None if it fails after all attempts
```

Indexer

The indexer.py file defines the Indexer class, which maintains an inverted index of URLs and text content. It indexes the URLs and the words found in the parsed HTML content, allowing for efficient search and retrieval of information.

Parser

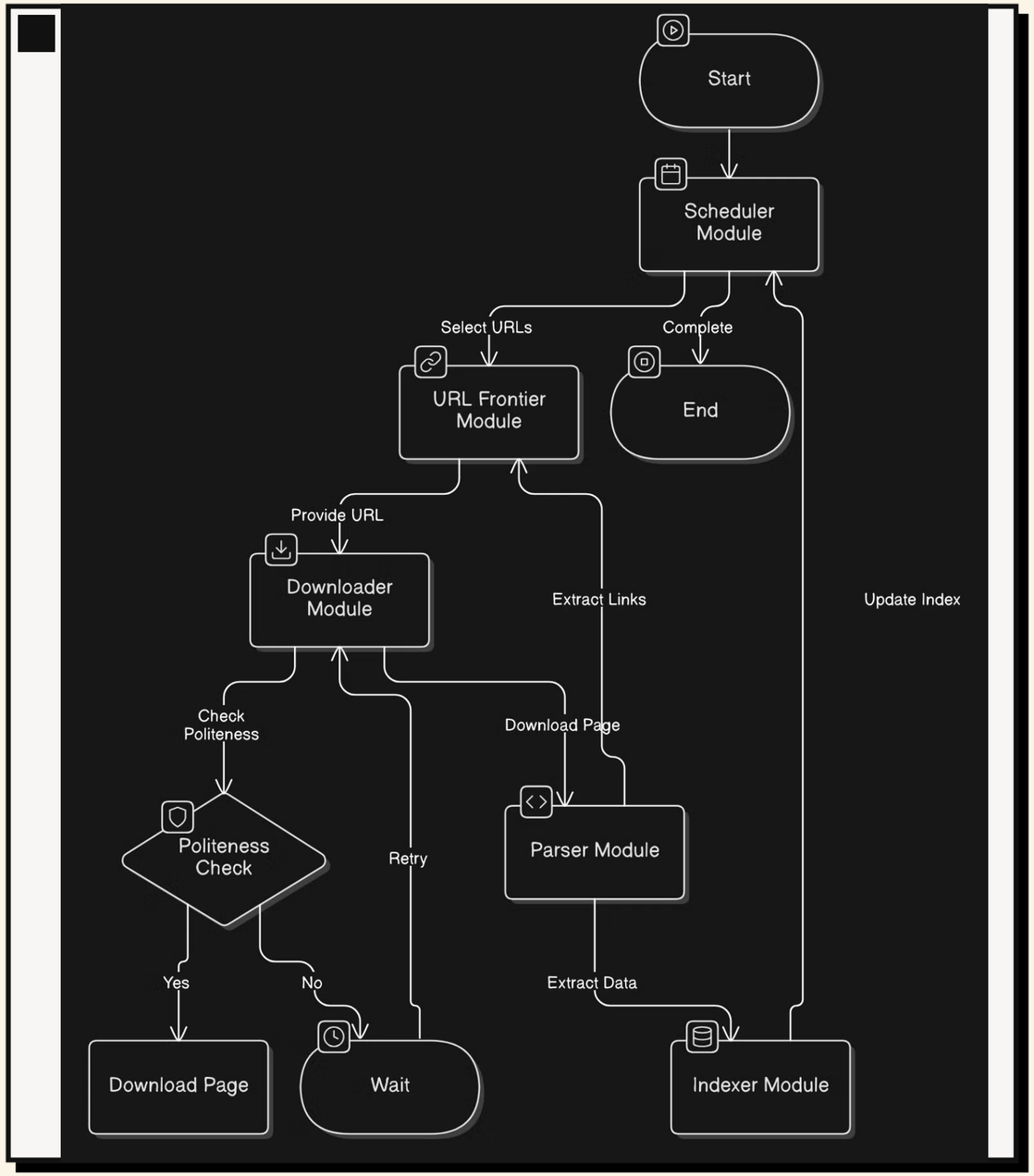
The parser.py file defines the Parser class, which is responsible for parsing HTML content. It extracts text, metadata, and links from the downloaded web pages. The parsed links are added to the URL frontier for further crawling, and the text and metadata are indexed for searchability.

Logger Config

The logger_config.py file defines the setup_logger function, which configures the logging for the application. It writes detailed log messages to a file and simplified log messages to the console, adapting the logging level based on the environment.

Robots.txt Handler

The robots_txt_handler.py file defines the RobotsTxtHandler class, which fetches and parses the robots.txt file of a website. It determines the crawling permissions for the user agent, specifying which paths are allowed or disallowed for crawling..


Start
Scheduler Module
Select URLs
Complete
URL Frontier Module
End
Provide URL
Downloader Module
Extract Links
Update Index
Check Politeness
Download Page
Politeness Check
Retry
Parser Module
Yes
No
Extract Data
Download Page
Wait
Indexer Module

Crawler's Time Line

Initialization

- The main.py script sets up the core components: URLFrontier, Downloader pool, Parser pool, Indexer, RobotsTxtHandler, and Scheduler.
- Logger configuration is initialized.

Crawling Process

- The main.py script handles command-line arguments to start crawling with a seed URL, depth, and robots.txt respect flag.
- The Scheduler coordinates the crawling process by managing the URL frontier, assigning tasks to downloaders, and forwarding downloaded content to parsers.

Scheduler

- Manages the overall crawling workflow, assigning tasks to downloaders and parsers, and ensuring the process adheres to depth limits and robots.txt rules.

RobotsTxt Handler

- Fetches and parses robots.txt files to determine crawling permissions for URLs.

Crawler's Time Line

Downloader Module

- Downloaders fetch HTML content from URLs provided by the Scheduler, adhering to politeness policies and retry mechanisms.
- Fetched content is forwarded to the parser queue.

Parser Module

- Parsers extract text, metadata, and links from the HTML content.
- Extracted links are added to the URL frontier, and the text and metadata are sent to the Indexer.

Indexer Module

- The Indexer maintains an inverted index of URLs and text content for efficient search and retrieval.

Completion

- Once all URLs are processed, the Scheduler signals the termination of the crawling process, and the results are printed or logged.

Ain't this a 10/10 ?