WEB CRAWLER

Kamil Milik

Wykorzystane Techniki:

W programie wykorzystuje:

- Klasę **Node**, która reprezentuje wierzchołek, posiada atrybuty wierzchołka: kolor, zawartość, kolejność odwiedzenia.
- Klasę **StringIntConverter**, która konwertuje tekst na liczbę. Ten zabieg przyśpiesza nieco pracę programu poprzez mniejsze wykorzystanie pamięci RAM, ponieważ Integer zajmuję mniej miejsca niż String.
- Klasę **EmailsAndLinksOperation**, która posiada metody:
    - deleteWrongLink – usuwa linki z niepoprawną końcówką, jak na przykład http://google
    - deletePNGFromEmail – usuwa maile które mają błędną końcówkę, jak example@.sth.png
    - searchEmailAndAddToSet – funkcja przeszukuje emaile na danej stronie, zlicza je , wypisuje na ekran i dodaje do zbioru
    - readSourceCodePage – służy do pobrania kodu źródłowego strony

Wykorzystane Algorytmy:

- **BFS**- klasa Bfs
- **DFS** – klasa Dfs

Użyte Struktury Danych:

- **Listy Sąsiedztwa** grafu – Klasa Graph, gdzie możemy dodać dany wierzchołek i jego sąsiadów, a także usunąć danego sąsiada. Używam HasMapy do zaimplementowania tej struktury.
- **HashMapa** – Klasa HashMap z podstawowymi operacjami hashmapy jak dodawanie, usuwanie, pobranie elementu.
- **Zbiór** – Klasa HashSet, która bazuje na klasie HashSet i ma podstawowe operacje, jak dodanie do zbioru oraz sprawdzenie czy dany element znajduje się w zbiorze.
- **Kolejka FIFO** – klasa QueueLinkedList, która posiada interfejs MyQueue. Klasa ta posiada podstawowe operacje kolejki FIFO jak: enqueue, dequeue oraz sprawdzenie czy kolejka jest pusta.

Uzasadnienie Zastosowania Konkretnych Rozwiązań:

Do przeszukiwania sieci użyłem dwóch algorytmów: DFS i BFS, użyłem ich ponieważ zapewniają, że każda strona zostanie odwiedzona tylko raz i przeszukana. Ponadto przeprowadziłem statystykę, który z algorytmów znajdzie więcej emaili i w jakim czasie. Oba algorytmy mają za zadanie przeszukać taką samą liczbę stron, czyli 100 stron. Wyniki prezentują się następująco:

DFS:

```
Number of bad addresses: 13. Number of e-mail addresses searched: 24
Total execution time: 88 seconds = 1 minutes 28 seconds
```

BFS:

```
Number of bad addresses: 8 NNumber of e-mail addresses searched 55
Total execution time: 144 seconds = 2 minutes 24 seconds
```

Wnioski:

DFS jest szybszy może tak być dlatego, że przegląda tylko jednego sąsiada w tym przypadku ponieważ, program się kończy po 100 stronach internetowych i przez to DFS nigdy nie wraca z rekurencji, żeby ponownie przeglądać resztę sąsiadów. Również implementacja może mieć wpływ na szybkość obu algorytmów.

Ciekawy jest fakt, że to **BFS zgromadził więcej adresów email**, widocznie zaczynając od strony, która miała dużo adresów email, przeszukuje wszystkie linki od tej strony prowadzące i widocznie te linki zawierają więcej adresów email, a co za tym idzie BFS w tym przypadku jest lepszy pod względem zebranych adresów.

Problem niepoprawnych linków rozwiązuje w algorytmie BFS tak, że wypisuje informację o złym linku, zliczam go, a także biorę kolejny link kolejki FIFO i aktualizuje ojca danego linku i wykonuje to tak długo aż link będzie działał prawidłowo.

Z kolei w algorytmie DFS, gdy napotkamy na niepoprawny link to również wypisuje go i zliczam, a także zmniejszam czas odwiedzenia, który tak naprawdę nie jest nigdzie używany, jednak chciałem zachować wzór tego algorytmu i postanowiłem zaimplementować czas odwiedzenia. Jest on zmniejszany ponieważ, gdy napotkam zły link to zwiększam go, a dopiero później sprawdzana jest jego poprawność, również usuwam go z listy wierzchołków oraz zmieniam aktualny wierzchołek na ojca, po czym wychodzę z danej rekurencji i sprawdzam następnego sąsiada danego ojca, który to ojciec posiadał nieprawidłowy link.