Trabalho Prático #1 - Web Crawler

JOÃO ANTONIO OLIVEIRA PEDROSA, Universidade Federal de Minas Gerais, Brasil

Esta documentação descreve um web crawler implementado com o objetivo de coletar um conjunto de páginas da web contendo 100.000 exemplares. O crawler coleta as páginas a partir de um conjunto de seeds e visita todos os links que encontrar no caminho, realizando a mesma coleta repetidamente até encontrar a quantidade requisitada de páginas.

Additional Key Words and Phrases: web crawler, python, paralelization

1 IMPLEMENTAÇÃO

A implementação foi realizada utilizando a linguagem Python em sua versão 3.8

1.1 Funções

O código do web crawler é relativamente simples e foi dividido em algumas funções. Todas essas funções se encontram devidamente documentadas no código em relação ao que recebem e retornam então documentarei aqui apenas uma breve explicação do funcionamento e qual a lógica usada para a implementação de cada uma.

1.1.1 `handler`. É feito o uso do módulo *signal* para tratar *timeouts* i.e. quando uma requisição demora tempo demais para ser completada e, portanto, é cancelada. O handler é uma função auxiliar que é passada para o módulo *signal* e explica o que deve ser feito caso o tempo marcado seja ultrapassado. No caso, o que é feito é o levantamento de uma *Exception* no Python. Dentro das outras funções, existe uma estrutura *Try - Except* que captura a exceção e, ao invés de parar a execução do código, simplesmente o executa novamente com argumentos diferentes. Dessa forma, páginas com um tempo de resposta muita altas não travam o crawler.

1.1.2 `get_html`. Recebe uma *url*, faz a requisição dessa url, armazena a resposta no devido arquivo WARC e retorna o conteúdo HTML da página.

1.1.3 `get_text`. Essa função é utilizada para a função de *debug* do crawler. Recebe um inteiro *n* e um objeto do tipo *soup* e retorna as *n* primeiras palavras do texto contido nesse objeto. Um objeto do tipo *soup* é uma instanciação de uma classe implementada pela biblioteca *BeautifulSoup4*.

1.1.4 `visit`. Essa é a única função do código que é executada com paralelismo. Após certas observações de performance foi percebido que o gargalo da aplicação se encontra no download e requisição das páginas Web. Sendo assim, para evitar conflitos de compartilhamento de variáveis entre *threads*, a função foi implementada de forma que a sua execução não afete a execução de outras threads. A função, portanto, recebe a url que deve ser visitada e retorna uma tupla contendo a url que foi visitada, o html contido naquele url e o url do protocolo robots daquele domínio.

1.1.5 `add_links`. Função que avalia quais links são elegíveis e os adiciona a fronteira de visitação do crawler. É recebido um objeto do tipo *soup*, um objeto do tipo *robots*. Com essas informações, a função analisa todos os elemntos do tipo *a* que possuem *href* e determina, baseando-se no protocolo robots, se esses links devem ser adicionados à fronteira de páginas a serem visitadas.

1.1.6 `print_debug`. Função auxiliar usada para imprimir o *debugging*. A função cria um dicionário com as informações necessárias e posteriormente usa o módulo *json* para fazer a impressão no formato correto.

Author's address: João Antonio Oliveira Pedrosa, joao.pedrosa@dcc.ufmg.br, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brasil.

1.1.7 `crawl`. Função principal da aplicação. Recebe um buffer com tuplas do tipo especificado no retorno da função *visit* e realiza o processamento de todas as páginas que se encontram nesse buffer. O fluxo da função é o seguinte:

Calcular a timestamp de acesso da página → Calcular domínio principal e os objetos *soup* e *robots* da página → Chamar print_debug → Chamar add_links → Adicionar timestamp ao dicionário de tempos de domínio e Url ao conjunto de Urls visitadas.

1.1.8 `main`. A função main da aplicação interpreta os argumentos, inicializa as estruturas e implementa um *loop* de preparação de *batches* para chamadas da função `crawl`. Os *batches* são implementados da seguinte maneira. Enquanto a quantidade de páginas vistiadas é menor do que o limite, a função *visit* é chamada em paralelo com o número de threads definido por uma variável de ambiente e com as primeiras páginas da fronteira de páginas a ser visitada. Os retornos dessas funções chamadas em paralelo são colocados em uma lista, chamada de *buffer* e esse buffer é enviado como argumento para a função crawl.

Uma questão que merece atenção especial é a explicação de como a escrita nos arquivos *WARC* é feita em paralelo sem o uso de Mutex. A função visit recebe, juntamente com uma url, uma id que indica em qual arquivo WARC deve ser escrito a resposta que for obtida. Um offset de em qual arquivo começar a escrever é mantido durante todo o processamento da função main, assim, independete de quantas páginas existam no buffer e quantas threads sejam chamadas, a escrita sempre é continuada a partir do último arquivo em que o loop anterior escrever. Dessa forma, todos os arquivos no fim recebem a mesma quantidade de páginas.

1.2 Estruturas de Dados

Algumas estruturas de dados são essenciais para o eficiente funcionamento do código e são variáveis globais acessadas por todas as funções. O nome das variáveis que as contemplam, a utilidade e a complexidade de cada uma é documentada na seção a seguir:

1.2.1 frontier. Estrutura do tipo Fila que contém as páginas a serem visitadas pelo crawler. Inserção, remoção e consulta em $O(1)$.

1.2.2 domains. Estrutura do tipo Dicionário que contém em cada elemento uma chave indicando um domínio e um valor indicando a última timestamp em que aquele domínio foi acessado. Esse dicionário é utilizado para manter uma distância de 100ms entre cada chamada para o mesmo domínio além de ser consultado para saber se um domínio já foi visitado anteriormente. Dicionários no Python são implementados com tabelas hash e, portanto, possuem inserção, remoção, edição e consulta em $O(1)$.

1.2.3 visited. Estrutura do tipo Set que contém o conjunto de Urls que já foi visitada. Assim como os dicionários, sets também são implementados com o uso de tabelas e portanto também possuem todas as suas operações em $O(1)$.

2 COMPLEXIDADE

Baseado nas complexidades das estruturas supracitadas e no fato de que todas são chamadas um número constante de vezes nas funções também citadas, pode-se afirmar que o programar tem um custo esperado amortizado de $O(1)$ por url, além do custo em percorrer, parsear e armazenar os textos contidos nas páginas visitadas. O custo computacional dessas operações, entretanto, não é especificado nas respectivas bibliotecas usadas para implementá-las mas, baseando se em implementações clássicas dessas tarefas, acho razoável assumir que tais operações executem com tempo linear na comprimento das strings processadas.

3 ESTATÍSTICAS

Corpus: Link para o Google Drive

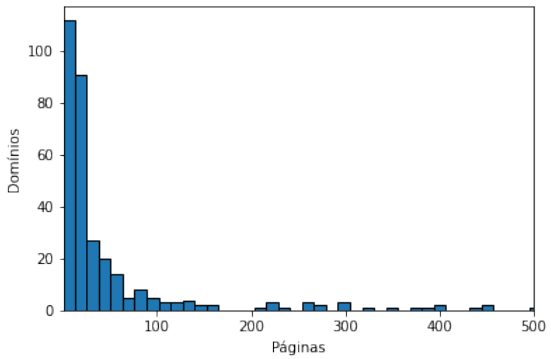
3.1 Estatísticas

Número de páginas visitadas: 100000
Número de domínios únicos: 341
Média de tokens por página: 442.13
Porcentagem dos 20 sites mais visitados da web atingida: 60%
Média de páginas visitadas por domínio: 6.70

Tabela com os 10 domínios mais visitados e o número de páginas visitada em cada dominio:

#	Domínio	Número de Páginas
1	ge.globo.com	12709
2	globoesporte.globo.com	10888
3	www.sportingnews.com	4094
4	www.nba.com	4030
5	thathi.com.br	3438
6	esporte.band.uol.com.br	2759
7	www.youtube.com	2508
8	www.band.uol.com.br	2483
9	sport.sky.it	2464
10	g1.globo.com	2168

Por existir uma variação muito grande no número de páginas visitadas por domínio, fica difícil exemplificar essa estatística para muitos domínios. Para tal, um histograma de número de paginas visitadas por número de domínios que visitaram essa quantidade de páginas é mostrado abaixo com o eixo X limitado no valor 500.



Número de páginas visitadas por domínio

Uma relação contendo todos os domínios e quantas páginas foram visitadas em cada um pode ser encontrada no arquivo *domains.csv*, na pasta que foi submetida juntamente com o código fonte da aplicação.