Trabalho de Recuperação de Informação - Web Crawler

DANIEL SOUZA DE CAMPOS, Departamento de Ciência da Computação - UFMG, Brasil

ACM Reference Format:
Daniel Souza de Campos. 2023. Trabalho de Recuperação de Informação - Web Crawler. 1, 1 (July 2023), 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

Um *Web Crawler* é a primeira parte do todo que forma um sistema de recuperação de informação. Ele deve ser capaz de seguir links recursivamente em páginas na Web com o objetivo de produzir um *Corpus*. Esse *Corpus* é formado por todas as páginas Web que o *Crawler* conseguiu rastrear.

Dado que a atual quantidade de páginas Web é desconhecida, mas estimada em bilhões, e também devido à natureza evolutiva de páginas já existentes, o trabalho de um *Web Crawler* não é simples. Em um sistema real, ele deve funcionar ininterruptamente e também deveria possuir uma infraestrutura capaz de suportar todo o abundante material coletado. Como já mencionado anteriormente, as páginas Web também são atualizadas a todo momento, com páginas sendo criadas, deletadas e modificadas. O coletor também deve tratar esses casos.

No presente trabalho, será apresentado o desenvolvimento de um *Web Crawler* com condições mais simples. Entre elas, ele deveria gerar um *Corpus* de 100 mil páginas, coletar estatísticas sobre elas e sem se preocupar com atualizações de páginas. Além disso, fatores como paralelização e políticas de rastreamento são discutidas.

2 FERRAMENTAS PARA O TRABALHO

O trabalho foi desenvolvido na linguagem Python 3.8.3 no editor de códigos *Visual Studio Code*. Para controle de versionamento e como repositório do trabalho foi utilizado o *GitHub* e o trabalho ficará disponível em https://github.com/Pendulun/WebCrawler após a data final de entrega do trabalho dia 12/05/2022. Lá será possível ver todos os *commits* realizados ao longo do desenvolvimento do programa. A máquina de desenvolvimento é um notebook DELL, 8GB RAM, 1TB HDD com processador Intel Core de sétima geração.

3 IMPLEMENTAÇÃO

Nesta seção será explicado qual é a estratégia geral de funcionamento do *Crawler* implementado.

Author's address: Daniel Souza de Campos, danielcampos@dcc.ufmg.br, Departamento de Ciência da Computação - UFMG, Belo Horizonte, Minas Gerais, Brasil.

XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3.1 Início

Um *Web Crawler* deve começar de algum lugar. Dessa forma, no início do trabalho foi disponibilizado um arquivo *seeds.txt* contendo 3 links para páginas consideradas *hubs* de links de onde o *Crawler* conseguiria iniciar o rastreamento. Uma observação importante é que uma das *seeds* era um link que não possuía nenhum conteúdo em sua página. Dessa forma, a classe *Crawler* é responsável por ler esse arquivo de entrada e iniciar o rastreamento a partir delas.

3.2 Pipeline entre Threads

Visando usufruir das capacidades de paralelização da máquina utilizada para o desenvolvimento e para melhorar a performance do programa como um todo, técnicas de paralelização foram implementadas. Uma boa observação é que a requisição de páginas web é uma tarefa demorada. Dessa forma, essa parte da tarefa se torna uma boa candidata como alvo de paralelização.

Visto que a requisição é uma operação de entrada e saída, o programa se torna *I/O bound*. Dessa forma, pode-se inferir que utilizar várias *Threads* já proveria um ganho de performance considerável.

Para concentrar as estruturas de dados compartilhadas pelas *Threads*, foi criada a classe *WorkersPipeline*. Ela possui a lógica de transferência de dados entre todos os trabalhadores, além de possuir mecanismos de parada das *Threads*.

3.3 Funcionamento de cada Thread

Uma instância da classe *Worker* representa todo o trabalho que uma única *Thread* separada deverá realizar. A principal ideia do programa é mapear links para *Threads* específicas por meio de uma função *hash* aplicada sobre o *host* da página. Dessa forma, *Threads* seriam responsáveis por requisitar páginas que tem como "dono" a própria *Thread*. Isso torna mais fácil o gerenciamento e respeito a políticas explicitadas no protocolo *Robots.txt* de cada *host*. O trabalho realizado por cada *Worker* funciona como o descrito nos próximos parágrafos.

Uma *Thread* espera ser alocado um link para fazer a requisição. Ao receber o link, é criada uma estrutura de dados para o host do link. Essa estrutura de dados é representada pela classe *Host* disponível no arquivo *Host.py*. Além disso, também existe uma classe para uma coleção de *Hosts* chamada *HostsInfo* disponível no mesmo arquivo. Importante ressaltar que a classe *HostsInfo* possui um mapeamento de *hosts* para objetos *Host* e cada um desses objetos possui um *set* de recursos a serem requisitados relativos àquele *host* e também um outro *set* de recursos já visitados. Se possível, a *Thread* acessa e salva as regras disponíveis no *Robots.txt* do *host*. Caso outro link futuro seja do mesmo *host*, suas regras já estarão salvas. Depois disso, caso seja o primeiro link de todos da *Thread*, ela coloca o *host* do link recebido em uma fila de prioridades ordenada pelo próximo *timestamp* em que será possível requisitar uma página de um *host* específico, ou seja, pares da forma (*timestamp:host*). No caso, como é a primeira vez do *host*, sua prioridade será 0, a máxima.

Feito isso, a *Thread* requisita à fila de prioridade de *hosts* o próximo *host* no qual ele poderá realizar uma requisição. Uma vez retornado, é verificado se o *timestamp* associado ao *host* está no futuro. Se sim, é esperado o tempo necessário para a requisição. Além disso, é requisitado à estrutura do *host* retornado um recurso a ser acessado. Podendo continuar, é feito uma requisição HEAD para o link com o intuito de analisar o seu *content-type*. Se esse for do tipo *text/html* e for retornado um código de status de algum tipo de sucesso, é feita uma outra requisição GET. Isso se faz necessário para a verificação rápida do tipo da página, já que é de interesse apenas páginas contendo texto e não apenas um vídeo por exemplo. Todo o acesso à internet é feito a partir da classe *WebAccess* disponível no arquivo *WebAccess.py*. Isso também se aplica à requisição pelo *robots.txt* do *host*.

Uma vez feita a requisição com sucesso, é realizado o *parsing* do conteúdo HTML retornado pela página. Toda *tag* HTML <a> é filtrada e o seu valor em *href* correspondente é salvo e tratado.

Já com todos os links de uma página salvos, é feita a distribuição desses para as suas respectivas *Threads*. Essa distribuição é feita aplicando uma função *hash* em cada um dos *hosts* dos links encontrados e os salvando em um dicionário mapeando os links às suas *Threads*. Aquelas que pertencerem à *Thread* atual serão imediatamente adicionadas à fila de prioridades como já foi descrito anteriormente.

Para aqueles que devem ser enviados a outras *Threads*, entra em trabalho a classe *WorkersPipeline*. Já nessa classe, cada link será mapeado à sua *Thread* específica em uma lista. Cada *Thread* possui a sua própria lista onde outras *Threads* poderão inserir links. Essas listas serão, eventualmente, consumidas pela *Thread* alvo. No caso, ficou definido que, a cada 15 requisições realizadas por uma *Thread*, ela deverá checar e consumir dessa estrutura.

Uma vez realizada a distribuição de links encontrados, a *Thread* atual salva o conteúdo da página requisitada em um arquivo de extensão *.warc.gz*. Esse processo é realizado na classe *WarcFileSaver* disponível no arquivo *WarcFileSave*.

Esse processo é realizado até que verificações de parada sejam atendidas. São elas: O máximo de páginas total estabelecido já foi requisitado e salvo com sucesso ou todas as *Threads* ficaram sem links e estão esperando outras *Threads* enviarem links para elas mas nenhuma delas depositou links na estrutura de distribuição de links. A primeira condição é mais fácil de acontecer, já que seria muito difícil nenhuma *Thread* ter links para requisitar eventualmente.

A Figura-1 mostra um resumo de como o sistema funciona.

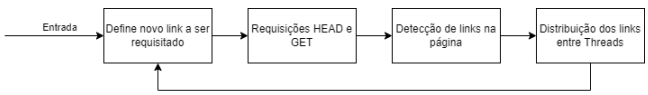

Fig. 1. Funcionamento simplificado do sistema implementado

4 CLASSES IMPLEMENTADAS

Essa seção tem como objetivo explicar, de forma geral, quais foram as classes implementadas e o seu objetivo.

4.1 WorkersPipeline

Disponível no arquivo *WorkersPipeline.py*. Essa classe é responsável por concentrar as estruturas compartilhadas entre as *Threads*. Dessa forma, ela contém estruturas de comunicação entre as *Threads* e os *Locks* responsáveis por garantir a sincronização do acesso às mesmas. Além disso, também é responsável por detectar quando o trabalho das *Threads* deverá terminar.

4.2 Worker

Disponível no arquivo *Worker.py*. Essa classe representa uma *Thread*. Ela é responsável por gerenciar as estruturas de dados relativas a uma única *Thread* e realizar as requisições para páginas.

4.3 UnwantedPagesHeuristics

Disponível no arquivo *Worker.py*. Essa classe tem como objetivo implementar heurísticas para ajudar a definir se um dado link poderá ser requisitado. Basicamente, ele possui um conjunto de extensões de páginas que devem ser evitados.

4.4 WebAccesser

Disponível no arquivo *WebAccesser.py*. Essa classe tem como objetivo encapsular todo o acesso à internet. É nela que, de fato, requisições para páginas são realizadas. Isso inclui o acesso ao arquivo *robots.txt*.

4.5 WarcSaver

Disponível no arquivo *WarcFileSave.py*. Essa classe implementa o método que salva informações de uma página em um arquivo de extensão *.warc.gz*. A cada 1000 páginas efetivamente salvas, um outro arquivo é usado.

4.6 HostInfo

Disponível no arquivo *Host.py*. Essa classe tem como objetivo encapsular todas as informações que cercam um único *host* como, por exemplo, as suas regras do *robots.txt*, recursos a serem requisitados e que já foram requisitados.

4.7 HostsInfo

Disponível no arquivo *Host.py*. Essa classe representa uma coleção de instâncias da classe *HostInfo*. Cada *Thread*/*Worker* possui sua própria instância de *HostsInfo*.

4.8 Parser

Disponível no arquivo *Parser.py*. A classe *Parser* é responsável por implementar métodos que realizam o *parse* de uma página HTML. Ela é capaz de recuperar todos os links contidos em *anchor tags*, além de formatar os links encontrados e recuperar as N primeiras palavras do texto visível na página.

4.9 DebugPrinter

Disponível no arquivo *DebugPrinter.py*. A classe *DebugPrinter* é responsável por imprimir na tela informações sobre a página que acabou de ser requisitada caso o sistema esteja em modo *debug*.

4.10 Arquivo utils

O arquivo *utils* implementa vários métodos relacionados a informações sobre um link. Por exemplo, ele implementa métodos capazes de separar um link entre host e recurso e também é nele que está implementado o método que retorna a qual *Thread* um link pertence.

5 DIFICULDADES E RESULTADOS

As maiores dificuldades foram com relação à paralelização em si do sistema. Da forma que foi implementado, muita sincronização teve que ser feita. Isso influenciou negativamente na performance programa.

Além disso, muitos problemas surgiram no desenvolvimento. O tratamento de exceções foi necessário em muitos pontos chave. Foi preciso muito tempo e testes até que esses pontos fossem identificados.

Também havia sido implementado uma política de *retries* de requisições para caso a internet caísse por algum tempo ao longo do *Crawling*. Isso teve que ser retirado pois existia um site específico que travava a requisição quando um *Retry* ocorria em seu site que estava em manutenção. O nome do culpado é https://www.threebit.io/. Cerca de três dias de trabalho foram investidos, até que esse único problema fosse identificado com a ajuda das ferramentas de possibilidade de análise de requisições do Mozilla Firefox.

No final, foi possível salvar 10412 páginas ao longo de uma hora e meia de execução com 80 *Threads*. O máximo de memória ocupada observada foi de cerca de 1,5GB. O resultado não foi o melhor esperado por causa da longa disputa entre programador e as dificuldades já mencionadas.

6 CONCLUSÃO

O trabalho tinha como objetivo desenvolver um *Web Crawler* que conseguisse seguir links em páginas web e formar um *Corpus* de 100 mil páginas. Pode-se dizer que, apesar de todo o esforço, esse objetivo não foi alcançado visto que o programa ainda contém problemas de sincronização na hora de término de operações e apenas 10412 páginas foram coletadas com sucesso.

Ao tentar desenvolver um *Web Crawler* foi possível aplicar os conceitos aprendidos em sala para essa primeira parte de um sistema de recuperação de informação. Além disso, também pôde-se perceber as dificuldades relacionadas ao *crawling* em si como inúmeras verificações de exceções relacionadas às requisições a serem realizadas e a efetiva implementação de uma estratégia de paralelização para melhorar a eficiência do sistema.

De forma geral, apesar dos resultados, houve um aprendizado enorme sobre o funcionamento de um *Web Crawler* e as suas dificuldades. Além disso, necessitou de muita criatividade ao pensar em uma estratégia de paralelização e em detalhes de implementação que poderiam fazer diferença na eficiência do sistema. Todos esses aprendizados com certeza serão levados em consideração para trabalhos futuros do aluno.