Lucas Chaves Lima

Recuperação de Informação - Máquinas de Busca na Web
Projeto e Implementação de um Crawler

Trabalho prático de Recuperação da Informação realizado na Universidade Federal de Minas Gerais com concentração na área de Ciência da Computação.

Universidade Federal de Minas Gerais – UFMG

Departamento de Ciência da Computação

Programa Pós-Graduação

Orientador: Berthier Ribeiro-Neto

Nivio Ziviani

Belo Horizonte

Abril de 2016

Sumário

1 Introdução

Recuperação de Informação (RI) é a atividade de recuperar itens de informação armazenados em um meio que possa ser acessado por computador. Após o surgimento da internet, a fim de facilitar o acesso do usuário à informação, surgiram mecanismos de busca. Mecanismos de busca são ferramentas capazes de procurar palavras-chave fornecidas pelo utilizador em documentos e bases de dados. A representação e organização dos itens de informação deve então prover ao usuário um acesso facilitado à informação de seu interesse. Esse mecanismo, pode ser dividido em três principais componentes: coletor, indexador e processador de consultas. A partir de um conjunto de páginas coletadas da Web deverá ser criado um índice para ser utilizado pelo processador de consultas. Esse trabalho terá foco na construção do coletor (Web Crawler).

1.1 Web Crawler

Um Web Crawler, também conhecido com Web Spider ou Web Robot, é um programa automatizado que percorre toda a World Wide Web. Geralmente, começam suas pesquisas em um site popular indexando as palavras encontradas na página a procura de novos links e assim, acessam os links encontrados e realizam o mesmo processo novamente. Frequentemente, web crawlers são utilizados para fazerem cópias de todas as páginas até então visitadas, páginas essas que serão processadas, indexadas e utilizadas posteriormente em alguma máquina de busca.

1.2 Objetivo

O objetivo principal desse trabalho é a implementação de um Web Crawler para coletas de páginas do domínio brasileiro. Para isso, começaremos a nossa coleta a partir de alguns sites populares (Seeds) previamente definidos e realizaremos a cópia de todas as páginas percorridas pelo Web Crawler. A visita as páginas deverá seguir uma fila de prioridade que será dada pelo nível das páginas.

1.3 Organização do Trabalho

O trabalho será disposto da seguinte maneira: Primeiramente apresentamos as Decisões de implementação e as principais tomadas de decisões, em seguida é apresentado a descrição do coletor implementado. Mais adiante é realizado a análise de complexidade do algoritmo. É realizado a análise dos resultados obtidos e apresentado a conclusão.

2 Decisões de Implementação

A seguir serão listadas as principais decisões que foram tomadas para a implementação do coletor de páginas brasileiras proposto para o TP1.

2.1 Principais decisões

- O coletor foi desenvolvido utilizando a linguagem C++, na versão C++11 2011 Standard.
- Foi utilizado a biblioteca Chilkat C/C++ Libs for 64-bit Linux para a realização do parser. Disponibilizada em: "http://www.chilkatsoft.com/chilkatLinux.asp".
- A coleta foi realizada apenas em páginas de domínio brasileiro.
- Foram utilizadas como seeds as seguintes páginas:
  1. http://www.catho.com.br/
  2. http://www.infojobs.com.br/
  3. http://vocesa.uol.com.br/
  4. http://www.zap.com.br/empregos
- A tomada de decisão do próximo link é realizado de acordo com o nível do link, decisão essa tomada por uma heap PriorityQueue.
- Respeita a regra de politeness, ou seja, não faz acesso a páginas de um mesmo domínio em um intervalo de tempo de 30 segundos.
- Foi implementado um crawler multi-threading.
- Buffer de páginas coletadas em Memória, a fim de aumentar a eficiência reduzindo as escritas em disco.
- Utilização de duas Queues para controles de acesso, PolitenessQueue e PriorityQueue.
- Controle de quantidade de links de um mesmo domínio na PolitenessQueue. Máximo de 10 links simultaneamente, evitando ociosidade das threads.
- Coleta de páginas utilizando 100 threads.
- Coleta de aproximadamente 2,7 milhões de páginas totalizando 190GB.

- A saída segue o seguinte formato:

  ||| URL | HTML DA PAGINA COLETADA |||.

2.2 Descrição do coletor Implementado

Abaixo é apresentado um diagrama do funcionamento das principais funcionalidades do Crawler implementado.

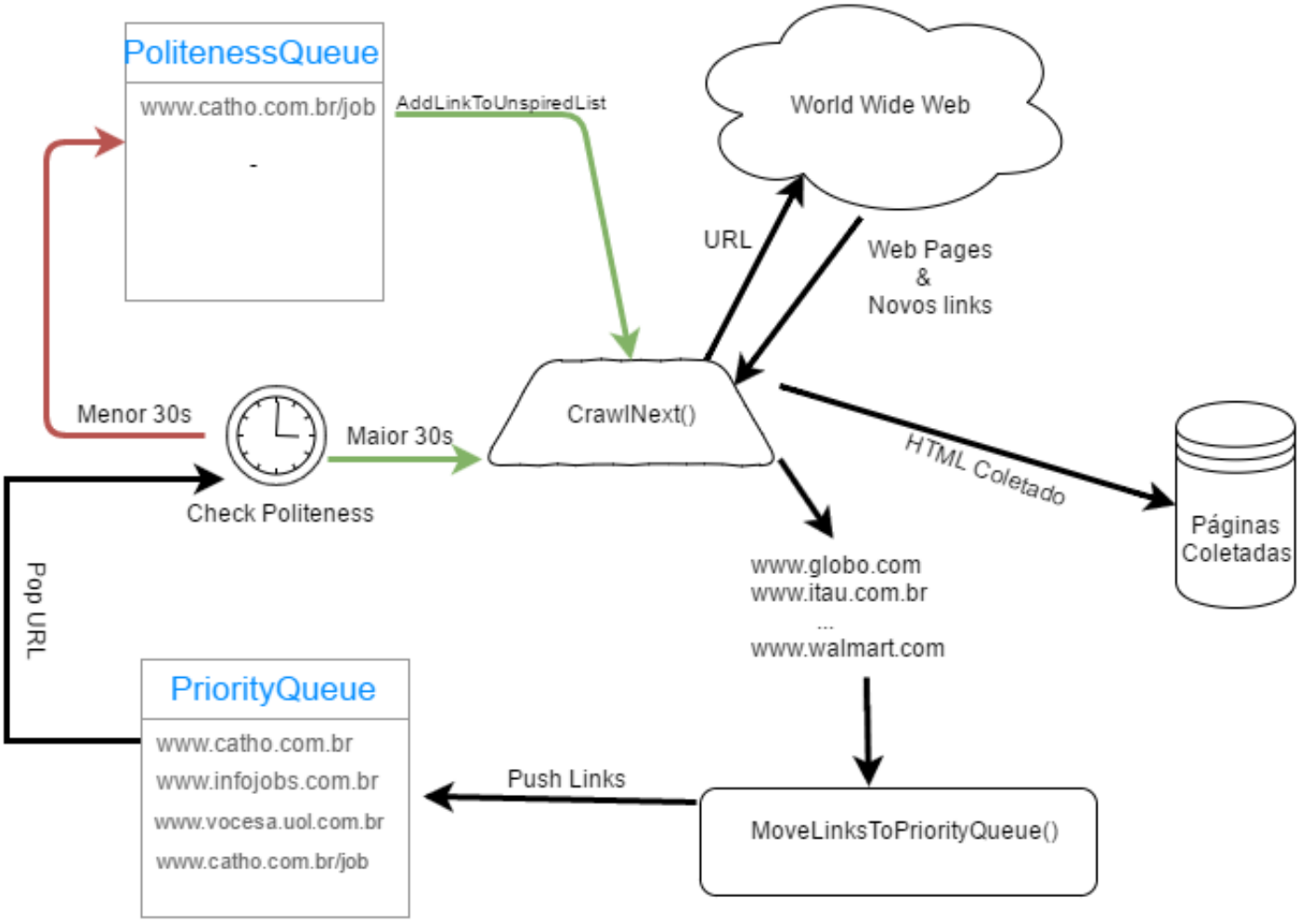

Figura 1 – Diagrama de funcionamento do Crawler

2.2.1 Estrutura do coletor

Foi criado uma classe chamada MyQueue, essa classe tem como principal objetivo controlar a Priority Queue, evitando condições de corrida entre as Threads e as condições de Politeness. As principais estruturas que ela possui são:

- Links - Struct que possui Url e seu respectivo level;

- PolitenessLink - Struct que possui um Links e um long contendo o tempo em que um domínio poderá ser acessado.
- PriorityQueue - É um vector de Links que controla qual deve ser o próximo link a ser coletado, a decisão é feita através do nível da url.
- PolitenessQueue - É um vector de PolitenessLinks que controla quando o próximo link de um mesmo domínio poderá ser acessado, a decisão é feita através do tempo de acesso.
- AlreadyAddedUrls - Hash de controle de links já coletado. Suas keys são os Urls, seu valor é definido True após uma Url nova ser adicionado a PriorityQueue.
- DomainHash - Hash que controla para cada domínio já visitado quando poderá ter outro acesso à aquele domínio. Suas keys são os Dominios e seus valores contém quando poderá ser visitado uma URL aquele domínio novamente.

Essa classe possui os seguintes métodos:

2.2.1.1 Myqueue::push()

O método push() somente efetuará o push, se e somente se, a Url ainda não se encontra na Hash AlreadyAddedUrls e se o tamanho da minha PriorityQueue for menor que 10k. Essas restrições evitam links repetidos, evitam links que se "afunilam"demais em um mesmo domínio, mantendo assim a PriorityQueue sempre com páginas relevantes e evita que a memória estoure.

2.2.1.2 Myqueue::pop()

O método pop() foi criada para ajudar no controle de condições de corrida, ou seja, não deixar que duas threads retirem o mesmo item ao mesmo tempo. Dessa forma, foi definido como região crítica a retirada de items da PriorityQueue e PolitenessQueue.

2.2.2 Parsing

Para agilizar a implementação, utilizei a biblioteca Chilkat para realizar o parsing das páginas. No entanto, essa biblioteca é fechada fazendo necessário a utilização de algumas estratégias para adequarmos tal biblioteca a nossa proposta.

Essa biblioteca possui um método CrawlNext() que realiza o parsing da página e armazena os urls encontrados em duas listas: UnspiredList e OutBoundLinks. No entanto, essas duas listas não possuem ordem de prioridade e possui links de páginas fora do domínio brasileiro. Para solucionar tais problemas, após chamar o método CrawlNext(), é chamado a função MoveLinksToPriorityQueue() que realiza o seguinte procedimento:

```
1- Para cada URL das listas UnspiredList e OutBoundLinks;
2- Verifico a ocorrência de ".br" ou "br." na URL;
3- Caso a ocorrência seja verdadeira, chamo a função getLevelUrl(URL);
4- Realizo o push(Link);
```

A função getLevelUrl() utilizada no procedimento acima realiza o cálculo do Nível do URL. O cálculo do nível é realizado da seguinte maneira:

```
1- Verifico qual é o UrlDomain e BaseDomain do URL;
2- Verifico a quantidade de ocorrência de "." no UrlDomain e BaseDomain e faço a dif
3- Verifico a quantidades de ocorrências de "/" na URL e multiplico por 10;
4- O resultado da soma dos passos 2 e 3 é o nível do meu URL;
5- Retorno a struct Links;
```

O nível foi calculado priorizando a qualidade de cobertura, ou seja, diversidade de domínios diferentes. Para fazer isso ponderei a ocorrência de "/", fazendo com que links que possuam barras tenham níveis mais altos, logo, ficarão no final da minha lista de prioridade.

2.2.3 Politeness

Para atender a condição imposta de Politeness de 30 segundos, implementei a estratégia de criar duas Queues: PriorityQueue e PolitenessQueue. A PolitenessQueue foi criada para receber Links que devem aguardar para serem acessados a fim de respeitar a condição Politeness. Essa Queue tem como função de prioridade o tempo de acesso, ou seja, até quando aquele URL adicionado deve ser impedido de ser acessado. Logo, a PolitenessQueue é constituído de um Links e um long que recebe quando o URL daquele link poderá ser acessado, o que chamamos de BlockedUntil. Quando é realizado o Pop(), se o primeiro item da PolitenessQueue possui tempo de BlockedUntil menor que o tempo atual, retira-se da PolitenessQueue, caso contrário retira-se da PriorityQueue. Quando retirado da PriorityQueue, atualiza-se a DomainHash colocando o tempo em que foi coletado mais 30s, dessa maneira, caso algum link do mesmo domínio for acessado em um intervalo de tempo menor que 30s ele será impedido realizar a coleta e será adicionado a PolitenessQueue. A PolitenessQueue possui apenas 10 URLS de um mesmo domínio simultaneamente, para evitar casos de termos apenas URLS de um mesmo domínio aguardando para ser coletado e as threads ficarem ociosas esperando 30segundos para acessar um novo link. 2-

2.2.4 Coleta

Para realizarmos a coleta temos a função Crawling(), cada uma das threads cria uma instância da função Crawling(). Primeiramente é declarado o CkSpider, que é uma classe da biblioteca Chilkat. Em seguida é criado um loop infinito (visto que não coletaremos todas as páginas, mas o máximo possível), dentro desse loop é feito o pop(). Primeiramente, a função pop() verifica se é permitida a retirada na PolitenessQueue, isto é, confere se o tempo atual é maior que o tempo BlockedUntil, esse procedimento nos garante a coleta de Urls que ainda não podiam ser acessadas pois estavam respeitando a Politeness, se essa condição é verdadeira, é retirado a URL a ser coletada da PolitenessQueue e adicionado essa URL a UnspiredList.

Na situação em que a condição anterior for falsa, o pop() será feito da PriorityQueue e em seguida é verificado na Hash DomainUrls se o domínio desta URL já pode ser acessado, se a condição de Politeness for satisfeita, é retornado o URL, caso contrário, adiciona-se a URL a PolitenessQueue e chama-se novamente o método pop(). Em seguida, é feito o CrawlNext() do URL retornado para fazer o parsing e verificar a existência de novos links, é chamada a função MoveLinksToPriorityQueue() para realizar a inserção dos novos links na PriorityQueue. Por último, adiciono à um Buffer o HTML da página coletada e o seu URL, quando esse buffer chegar a um limite estabelecido, é realizada a escrita em arquivo.

2.2.5 Observações sobre Biblioteca

Apesar de facilitar no parsing das páginas, por se tratar de uma biblioteca fechada, encontrei algumas dificuldades enquanto utilizando essa biblioteca. Primeiramente, há a necessidade de realizar o Initializer para cada página, isso é um processo mais lento do que apenas dar o CrawlNext(). Um outro problema, foi quando utilizei o split da CkString já implementado na biblioteca, pois por algum motivo estava dando overflow de memória. Quando é feito o CrawlNext() ele cria as duas listas Unspired e OutBound automaticamente, o que fazia ficar mais lento por ter que percorrer cada um dos itens dessa lista adcionando a priority e depois retornando a Unspired. Por fim, um dos maiores motivos da entrega atrasada, foi que eu estava com um problema que após aproximadamente uns 150k de páginas coletadas, ficava muito lento o processo de coleta. Consegui descobrir que o problema estava no CkSpider, então, para agilizar o processo após umas 150 páginas coletadas pela thread, eu deleto o spider e o crio novamente, solução que resolveu o problema. Aconselho a biblioteca, mas não para utilizar de forma personalizada.

2.3 Análise de complexidade

Visto que foi utilizado uma biblioteca fechada, não consigo dizer ao certo qual é a complexidade do Parsing, no entanto, listarei aqui as complexidades das funcionalidades implementadas. Na próxima subceção descrevemos a análise de complexidade, a tabela abaixo apresenta alguns símbolos utilizados na análise.

Tabela 1 – Definição dos símbolos comuns.

Símbolo	**Descrição**
C	Número de urls coletados.
N	Número de links existentes na lista.
K	Tamanho da Url
H	Tamanho da Heap

Após inserida as Urls nas listas OutboundLinks e UnspiredLinks, a função MoveLinksToPriorityQueue tem de percorrer todos os items de ambas as listas, logo a complexidade disso é 2*O(N). Também nessa função é verificado se os links são do domínio .Br, essa verificação é O(K) * N.

A função de verificação do nível de um URL é feita através da contagem de ocorrências de caracteres na URL, como é verificado duas vezes para cada url (ocorrência de . e /) a complexidade dessa função é O(K).

Como a PriorityQueue e PolitenessQueue são heaps, o pior caso de inserção é O(C logC). Pode-se observar que a complexidade de inserção vai aumentando a medida que o tamanho da Heap cresce, no entanto a a heap tem fixada tamanho máximo de 10 mil, logo o seu pior caso será O(10000 * log 10000). Limitando o tamanho da Heap consegui estabilizar a velocidade de coleta.

Alguns métodos incluídos na biblioteca Chilkat são utilizados nessa função, logo não citarei ao certo a complexidade dos mesmos. A função principal do programa é a Crawling(), essa função é executada por múltiplas threads. Primeiramente a função realiza o Pop() da PriorityQueue logo a complexidade O(1), em seguida é realizado o Crawlnext() do qual a complexidade é desconhecida, feito isso é chamada a função MoveLinksToPriorityQueue para passar os links para PriorityQueue, logo complexidade (2*O(N) + O(K) * N). Para o cálculo de nível do URL complexidade O(K) * número de links inseridos na PriorityHeap logo O(K) * C. E a complexidade de inserir na heap (C log C). Temos assim, por fim a complexidade geral do algoritmo que é ((2*O(N) + 2*O(K) +O(K)* N) + "Crawlnext() Complexidade"+ log C) * C).

3 Análise de Resultados

As páginas foram coletadas utilizando o computador Intel(R) Core(TM) i5-3350P CPU @ 3.10GHz, com 16gb de memória ram, sob a plataforma Ubuntu 14.04, onde o software foi executado via Terminal. A conexão utilizada foi a do laboratório LATIN, não sabendo ao certo qual é a banda total.

3.1 Páginas Coletadas

A coleta foi efetuada no computador especificado acima. Abaixo segue uma tabela mostrando os resultados obtidos:

Tabela 2 – Resultados obtidos da coleta

Número de páginas coletadas	2,5milhões
Threads utilizadas	100
Tempo de coleta	29 horas
Tamanho da coleção	190 GB
Velocidade média da coleta	23,88 Pág/s
Velocidade média de download	1,85 MB/s

Como pode ser observado na tabela acima, o coletor manteve uma média de 23,88 páginas por segundo, o que é considerado bom se comparado a um coletor industrial que coleta em média 1 milhão de páginas por dia, enquanto o coletor implementado coletou 1 milhão de página em aproximadamente 6,67 horas. As coletas estão abrangendo uma grande variedade de domínios e não se afunila em apenas um domínio, ou seja, possui uma boa vizinhança. Por exemplo, em uma coleta de 1000 páginas, aproximadamente 500 dessas páginas coletadas são de domínios diferentes, ou seja, em média 50% das páginas coletadas. Além disso, como adiciono apenas 5 páginas do mesmo domínio por vez na minha PolitenessQueue, o controle de boa vizinhança é mantido ao longo de todo a coleta.

Apesar da média de páginas por segundo ter sido 23,88, uma obervação feita foi de que a partir de uma certa quantidade de páginas coletadas, o coletor passa a fazer a coleta de maneira mais lenta, ou seja, há um decaimento na taxa de páginas por segundo. Até 1 milhão de páginas, não houve decaimento, apenas algumas variações onde a média se manteve em aproximadamente 40 páginas por segundo, no entanto, a partir de 1 milhão

de páginas a taxa de transferência caiu chegando a um mínimo de 8 páginas por segundo. Essas informações podem ser melhores observadas no gráfico abaixo:

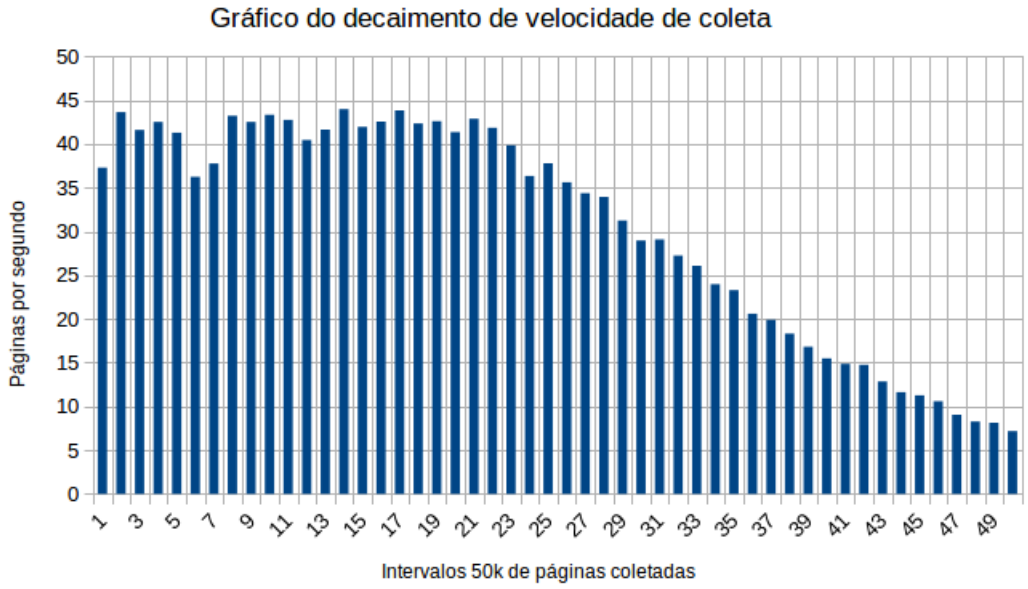

Figura 2 – Decaimento da velocidade de download de páginas

Abaixo é apresentado um gráfico mostrando o Speedup Multi-Threading.

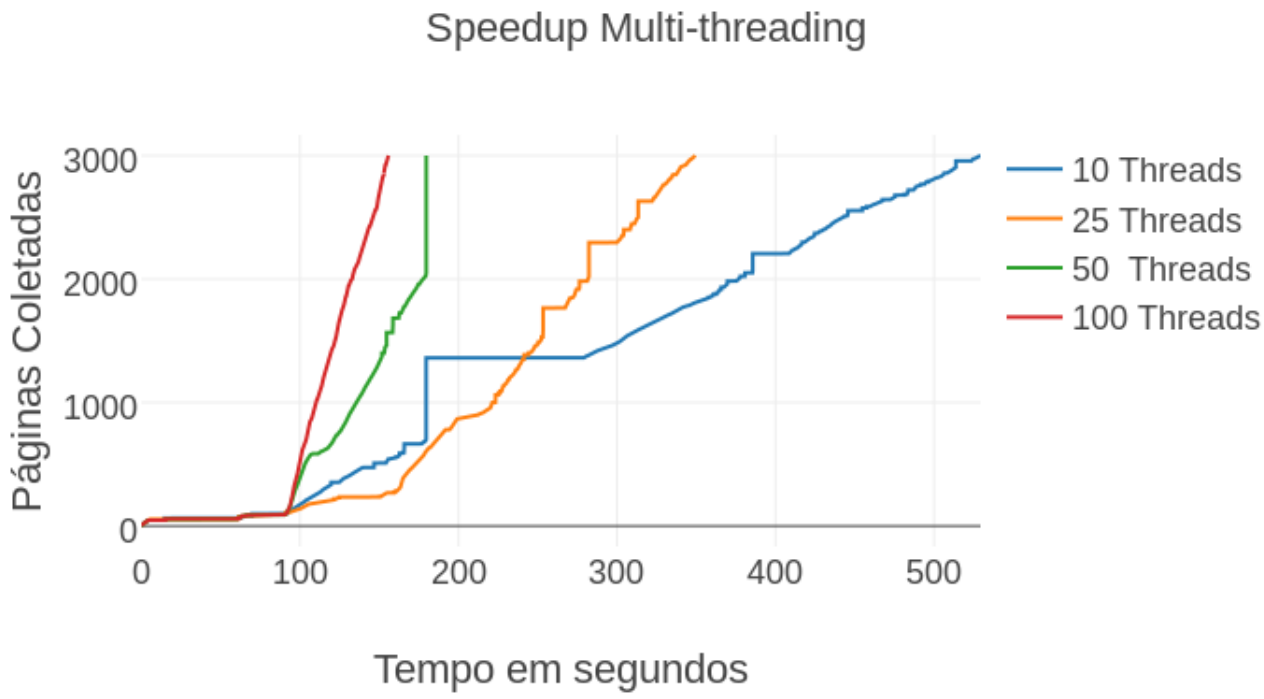

Figura 3 – Variação da velocidade multi-threading

Em relação a implementação de multi-threads, pode-se observar no gráfico acima que não foi obtido um speedup linear. No entanto, quando é realizado a coleta por 10

minutos utilizando 10 e 100 threads, a quantidade de páginas coletadas é passada de 3438 para 25000, ou seja, é 7,35 vezes mais rápido. Conclui-se assim que o speedup é aproximadamente 3/4 * NumThreads. Para testar o speedup obtido pelo multi-threading, realizei um experimento calculando tempo necessário para a coleta de 3000 páginas. Fiz uma verificação variando o número de threads em 10, 25, 50 e 100.

4 Conclusão

Esse trabalho teve como principal objetivo a implementação de um coletor para páginas do domínio brasileiro. Com o desenvolvimento dessa etapa de coleção, foi possível aprender ainda mais sobre o funcionamento de um Web Crawler e como é importante a utilização de boas técnicas de programação e otimização em códigos quando trabalhando com grandes volumes de dados. Este trabalho cumpriu como objetivo proposto, de colocar o aluno a frente de um Web Crawler, não só implementando, mas também avaliando as possíveis melhorias.

Obtivemos como resultado uma coleta de 2,5 milhões de páginas, com uma taxa de download de páginas média de 23,8pag/s, resultando em uma coleção de 190gb. Pode-se considerar como ponto forte a velocidade de download de páginas e a cobertura, visto que o coletor coleta 1 milhão de páginas em aproximadamente 6 horas e que foi implementado boas regras de vizinhanças. Acredito que um dos pontos fracos que pode ser observado no trabalho é há um decaimento da taxa de transferência após um grande número de páginas coletadas, no entanto, esse decaimento fica estável em uma taxa aceitável de download. Uma outra estratégia que poderia ter sido adotada, é ao invés de ignorar links quando a minha Queue estiver cheia, seria salvar esses links e escreve-los em disco, e recupera-los caso a queue fique vazia, no entanto, como havia um limite de tempo para a coleta, não acredito que havia a necessidade para esse trabalho. Algumas melhorias que poderiam ser consideradas nesse trabalho são:

- Utilizar ou implementar própria ferramenta para realizar parsing
- Melhorar o decaimento após grande quantidade de páginas coletadas
- Armazenar links que não forem inseridos na Queue e salva-los em disco para necessidades futuras.

Por fim, o trabalho realizado foi de imenso aprendizado para um aluno que pretende seguir a área de recuperação de informação.