Centro Federal de Educação Tecnológica de Minas Gerais

Departamento de Computação

Curso de Engenharia de Computação

Josué Rocha Lima

Tulio Fonseca Coqueiro

Caio Gonçalves Silva

Trabalho Prático: Implementação de um Coletor Web

Coletor Web

Relatório do trabalho prático apresentado à disciplina de Recuperação de Informação.

Orientador: Daniel Hasan Dalip

Belo Horizonte

2017

Sumário

1 Introdução

Nos últimos anos, a internet está em alto crescimento, impulsionado por uma maior adoção às tecnologias e, principalmente, pela popularização dos dispositivos móveis. Atividades que manipulam a Web se tornam cada vez mais repetitivas e cansativas, se feitas de forma manual. Atividades como: buscar informações, indexar paginas novas e salvar arquivos e imagens podem se tornar impraticáveis devido ao grande volume de dados na internet.

1.1 Contexto do trabalho prático

O *site* World Wide Web Size[1] disponibiliza gráficos que demonstram o crescimento da Web com base nos dados da Google e Bing. No mês de Setembro de 2017 a estimativa chegou perto de 52 bilhões de webpages existentes (Figura 1).

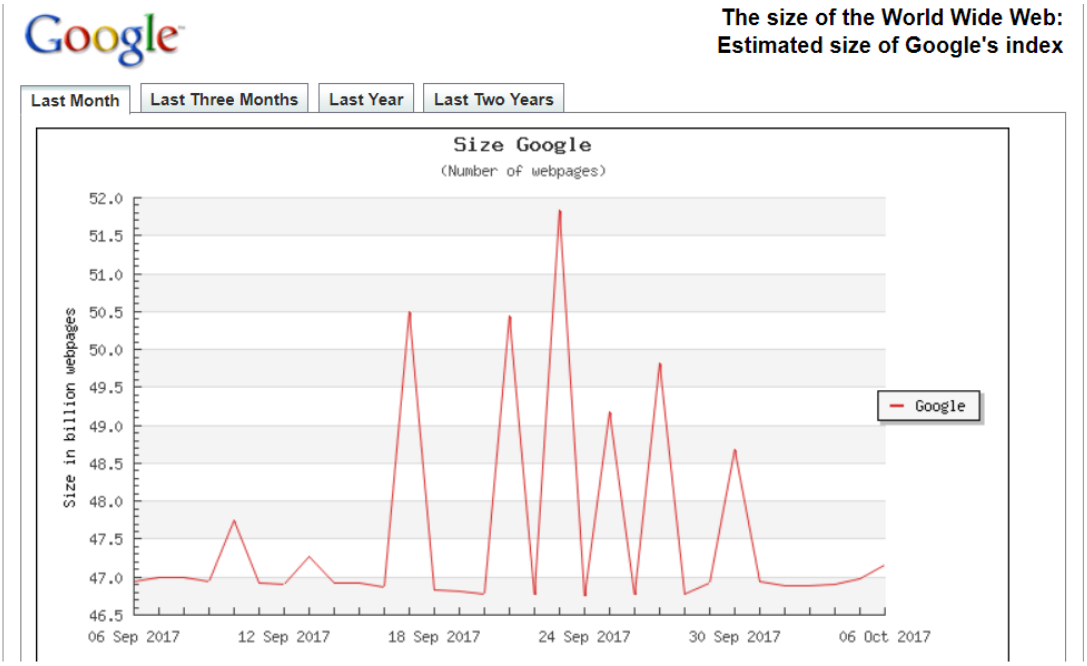

Figura 1 – Estimativa do tamanho da Web no mês de Setembro, segundo dados da Google. (KUNDER, 2017)

Devido a essa imensa quantidade, visitar manualmente página por página buscando por uma informação é um trabalho cansativo e ineficiente. Para resolver essa situação, é necessário um programa que automatize a busca na web, o Web Crawler. Atualmente, se tornou comum a utilização destes sistemas autômatos em empresas que necessitam buscar alguma informação na internet.

[1] <www.worldwidewebsize.com>

O Web crawler (web spider ou web robot) pode ser traduzido como "rastreador web", ou seja, é um programa que navega pela internet de uma forma metódica e automatizada, coletando websites para prover buscas futuras mais rápidas.

No geral, se começa com uma lista de URLs para visitar (sementes ou seeds) e à medida que o crawler visita essas URLs, identifica-se todos os links da página e os adiciona na fila de URLs para visitar. O Escalonador retira uma URL da fila e baixa seu conteúdo, salvando o texto e os metadados em um sistema de armazenamento (arquivo, memória, nuvem, etc).

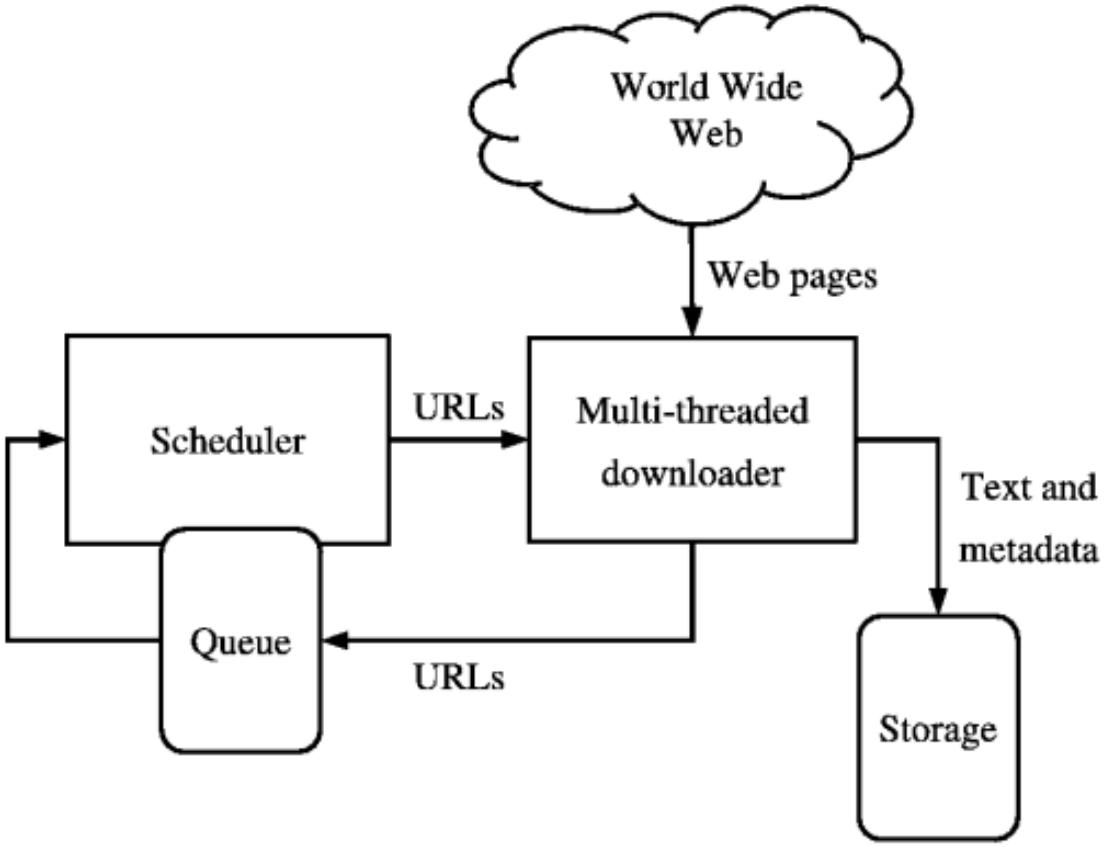

Figura 2 – Diagrama explicativo sobre o WebCrawler.

Grandes empresas já possuem seus próprios Web Crawlers:

- **Yahoo! Sluro** é o nome do Crawler do Yahoo!;
- **Msnbot** é o nome do Crawler do Bing – Microsoft;
- **Googlebot** é o nome do Crawler do Google.

Com o objetivo de tornar o crawler mais eficiente e preciso, segue-se uma combinação de políticas:

- **Política de Seleção** Como visto anteriormente, coletar a web completamente é uma tarefa impossível , os crawlers coletam somente uma parte dela. Por esse motivo, é desejável que essa pequena parcela contenha informações pertinentes à consulta.
- **Política de Revisitação** Como a web está em constante mudança, é necessário revisitar as paginas ja coletadas para atualiza-las. A frequência de revisitação pode ser uniforme, todas as paginas com a mesma frequência, ou proporcional, maior frequência para paginas que atualizam mais rápido.
- **Política de Cordialidade** É necessário respeitar o tempo de acesso a um servidor, para não sobrecarregá-lo. A frequência de acesso não pode ser muito alta.
- **Política de Paralelização** Para maximizar a taxa de download e otimizar a coleta, o crawler pode utilizar múltiplas threads simultâneas.

Além disso, há o Protocolo de Exclusão de Robôs (robots.txt protocol). É um padrão que indica para o coletor quais partes de um servidor podem ser acessados, funcionando como um filtro. O uso de robots.txt é uma maneira efetiva de se tratar permissões para os coletores e de evitar a sobrecarga em um servidor.

1.2 Trabalho

O projeto proposto pelo professor tem como base a implementação de um coletor simples para Web, com o objetivo de estudar e aprender sua arquitetura. O coletor tem que obedecer, obrigatoriamente, os protocolos de exclusão de robôs:

- critérios pertencentes no robots.txt;
- critérios “noindex” e “nofolow” das metatags de cada html extraído;
- prazo de, no mínimo, 30 segundos entre requisições em um mesmo servidor.

Os parâmetros utilizados no trabalho, foram:

- Número máximo de páginas para coleta = 500 páginas;
- Máxima profundidade por domínio = 4 páginas;
- Número de threads utilizados = 15 x número de núcleos do processador.

2 Metodologia

O presente projeto consistiu no desenvolvimento de um coletor de páginas web de propósito geral. Este projeto teve como objetivo a coleta de 500 páginas. Portanto, o projeto se fundamenta nas políticas de exclusão de robôs e boas maneiras de modo à evitar prejuízo e sobrecarga para a infraestrutura existente nos servidores alvo. Para iniciar a coleta começa com algumas URLs para serem visitadas (sementes ou seeds).
Nosso projeto teve como sementes as seguintes URLs:

- **Cable News Network (CNN)** <www.cnn.com>;
- **GQ Australia** <www.gq.com.au>;
- **HuffPost Brasil** <www.huffingtonpost.com>.

2.1 Políticas e padrões

A coleta foi realizada pautada nas políticas de exclusão de robôs e de boas maneiras visando realizar a coleta sem prejudicar os serviços oferecidos pela Web ou seus usuários. Consoante a isso, para possibilitar a identificação o coletor desenvolvido adota uma política padronizada de identificação.

Portanto, antes de efetuar coleta em um servidor, o arquivo *robots.txt* correspondente é sempre consultado e verificada a permissão de coleta na respectiva página por meio de análise das diretivas "Allow", "Disallow"e de *User Agents* permitidos. Após obtenção de permissão e coleta de uma página, *meta tags* destinadas à coletores são buscadas e é verificado se a página autoriza indexação (diretivas "index"e "noindex") e extração de links para posterior coleta(diretivas "follow"e "nofollow").

Conforme especificado pela política de boas maneiras, foi implementada uma pausa de 30 segundos entre coletas em um mesmo servidor. Deste modo, ao encontrar uma página coletada há menos de 30 segundos, o escalonador mantém esta página na fila e analisa a elegibilidade da próxima página da fila. Além disso, a profundidade das páginas recuperadas pelo coletor foi restringida a 4 camadas, visando efetuar coleta em largura e evitar muitas requisições sobre um mesmo servidor.

O coletor Web desenvolvido foi nomeado BrutusBot. Ao recuperar páginas, BrutusBot se identifica enviando o *User Agent* com seu identificador e página Web "BrutusBot (http://josuerocha.com.br/infoBrutusBot.html)", tornando possível o contato por parte dos administradores das páginas requeridas. As coletas efetuadas por BrutusBot têm propósito

estritamente didático. Identificação, informações adicionais e um e-mail de contato do BrutusBot podem ser consultadas na página <http://josuerocha.com.br/infoBrutusBot.html>. A Figura 3 ilustra a página Web do BrutusBot.

Figura 3 – Conteúdo parcial da página do coletor desenvolvido (<http://josuerocha.com.br/infoBrutusBot.html>).

2.2 Arquitetura do coletor web

A aplicação foi desenvolvida na linguagem de programação Java, adotando uma abordagem paralela para possibilitar requisição simultânea de documentos. Um diagrama de classes foi construído (Figura 4) para possibilitar melhor entendimento da aplicação.

A classe *PageFetcher* é a classe paralelizada e um objeto da classe EscalonadorSimples é compartilhado entre todas as instâncias de *PageFetcher*, atuando como centralizador e gerenciador das coletas. Pode-se observar a especificação das classes e métodos do sistema com maior nível de detalhe a seguir:

- **EscalonadorSimples**
  Objeto compartilhado por todas as *threads* que contem informações de controle como: fila de páginas visitadas, mapa ligando dado domínio à seu respectivo Record, um *Set* de páginas visitadas e uma lista de páginas coletadas.
  - **public EscalonadorSimples(String[] seeds)**
    Construtor da classe, adiciona cada URL presente no vetor seeds à fila de páginas.
  - **public synchronized URLAddress getURL()**
    Retira uma URL da fila cujo respeitando a política de boas maneiras e a retorna à *thread* requisitante se a coleta não estiver finalizada.
  - **public synchronized boolean adicionaNovaPagina(URLAddress urlAdd)**
    Adiciona a URL à fila de páginas caso a mesma não tenha sido visitada e tenha

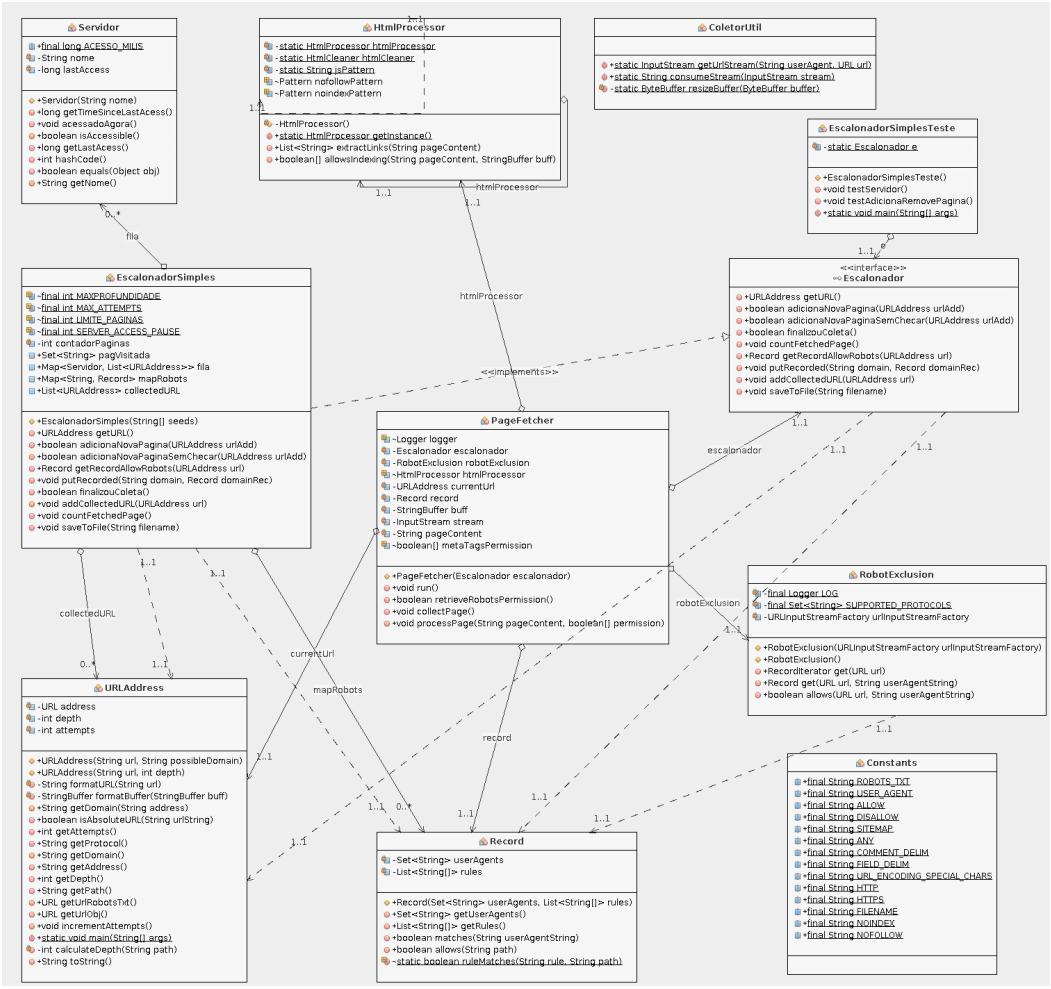

Figura 4 – Digrama de classes do coletor BrutusBot.

profundidade menor do que o valor especificado. Nessa implementação o valor 4 foi definido como profundidade máxima.

– **public synchronized boolean adicionaNovaPaginaSemChecar(URLAddress urlAdd)**
Adiciona a URL à fila de páginas e ao mesmo tempo remove-a do *Set* de páginas visitadas. Este método foi implementado para viabilizar novas tentativas para requisições não respondidas devido à erros de conexão. Essa abordagem possibilita aguardar até que um eventual problema na infraestrutura de rede seja resolvido enquanto outras páginas são coletadas pela *thread*. Um limite máximo de quatro tentativas foi estabelecido.

– **public synchronized void saveToFile(String filename)**
Salva a lista de URLs em um arquivo de texto em disco. A função é executada após a conclusão da coleta.

- **PageFetcher**
A classe representa cada thread, que por sua vez irá coletar as paginas e verificar as

permissões e insere novas paginas no escalonador.

- **public PageFetcher(Escalonador escalonador)**
  Cria um robot e um buff para dispensar a necessidade de criar outro toda hora. O contrutor tambem coloca um objeto para receber o escalonar para poder utiliza-lo na Classe.
- **public void run()**
  Será o ponto de partida da thread, ele atua como o pipeline geral da coleta, coletando a pagina logo apos coletando as metaTags, e por fim olhando as permissões de coletar/extrair os links das páginas.
- **public boolean retrieveRobotsPermission() throws Exception**
  Verifica as permissões dos Robots.txt. Caso já tenha as permissões já salva no escalonador, ele já acessa e verifica as permissões disponíveis. Caso não tenha, ele vai mandar a requisição pedindo pelo Robots.txt do servidor.
- **public void collectPage() throws Exception**
  Coletar a página.
- **public void processPage(String pageContent, boolean[] permission)**
  Verifica as permissões de coleta e de extrações de links, caso tenha ele realiza a extração/coleta.

- **HtmlProcessor**
  Ao receber toda a página, limpa todo o código HTML recebido encontrando os links e permissões para a coleta/indexação no formato de *meta tags* por meio de uma API especializada (HTMLCleaner).

  - **private HtmlProcessor()**
    Contrutor da classe, inicializa a API e configura a biblioteca. O construtor possui modificador de acesso privado para evitar que vários objetos da classe sejam instanciados.
  - **public static HtmlProcessor getInstance()**
    Implementa o padrão de projeto *Singleton* permitindo que um único objeto da classe *HtmlProcessor* seja utilizado por todas as *threads*, contribuindo para o desempenho computacional.
  - **public List<String> extractLinks(String pageContent)**
    Processa todas as tags de uma página, selecionando as tags "a"e extraindo o atributo "href". A cadeia extraída corresponde aos links presentes no documento. Uma lista contendo os links é retornada.
  - **public boolean[] allowsIndexing(String pageContent)**
    Processa todas as tags da página, encontrando todas as tags "meta", cujo

atributo "name"corresponde à "robots". O valor especificado pelo atributo "content"é processado para verificar se a tag possui a especificação NOINDEX e/ou NOFOLLOW para assim decidir as permissões de acesso a página.

- **PrintColor**
  Classe que contem os ANSI escape sequences para a mudança de cor da saída no terminal.

- **URLAddress**
  Contém um objeto do tipo URL, possui métodos para obter varios dados, como domínio, protocolo, profundidade, tentativas de requisição.
  Os métodos Get e Set foram omitidos, com o intuito de facilitar a leitura do relatório.

  – **public URLAddress(String url, String possibleDomain)**:
  Construtor da classe, inicialmente verifica se no final da URL existe apenas uma "/", caso exista, é retirada e a URL é adicionada ao StringBuffer. Essa verificação foi feita pois alguns sites possuem *links* contendo somente o caractere "/", gerando o site original após a concatenação com o domínio. Entretanto o site não era reconhecido como já coletado devido ao caractere adicional. Além disso, verifica-se a existência de "//"no início da URL, para inserido no inicio da mesma a especificação de protocolo "http:"ao invés de "http://". Caso a verificação acima for falsa, é verificado se a URL é absoluta,caso ela não seja domínio do servidor é concatenado ao *StringBuffer*. Por fim, é criado criado um objeto URL com a cadeia obtida e computada a profundidade da URL no servidor.

  – **public URLAddress(String url, int depth)**
  Construtor que permite especificar a profundidade da URL diretamente.

  – **private String formatURL(String url)**
  Formata a URL do link concatenando a string "http://"à URL.

  – **private StringBuffer formatBuffer(StringBuffer buff)**
  Formata a URL do link porém recebendo um objeto *StringBuffer*, foi criado com o intuito de otimizar o processamento, para evitar a instanciação de um objeto extra na hora de realizar a concatenação das *strings*.

  – **public String getDomain(String address) throws MalformedURLException**
  Retorna o servidor de domínio do endereço fornecido.

  – **public boolean isAbsoluteURL(String urlString)**
  Verifica se uma URL é relativa ou absoluta.

  – **private int calculateDepth(String path)**
  Calcula a profundidade de um link dado um caminho extraído da URL completa.

2.3 Principais desafios

Indubitavelmente o presente projeto contribuiu consideravelmente para o aprendizado. Entretanto, durante o desenvolvimento deparou-se com alguns desafios de implementação, tratamento de exceções e processamento de texto.

Durante a depuração e execução preliminar da aplicação, percebeu-se que algumas URLs coletadas não estavam sendo salvas no arquivo de texto de saída e inicialmente pensou-se que se tratava de algum erro de programação ou alguma particularidade de programação paralela.

Seguindo orientações do professor, foi utilizada a API HTMLCleaner, com a finalidade de extrair os links de páginas HTML mal-formadas. Porém, tornou-se necessário um estudo sobre a API em seu site oficial [1] e foram recordados recursos e tags da linguagem de marcação HTML (HyperText Markup Language).

O processamento de URLs gerou algumas objeções devido às representações utilizadas nas páginas Web. Muitos sítios Web especificam links absolutos no formato "//dominio/caminho", outros especificam "dominio/caminho"sendo assim necessário analisar o formato da URL antes de inserir o indentificador de protocolo com duas barras ou nenhuma (http:// ou http). Além disso, links relativos para a mesma página são especificados por somente uma "/"em alguns casos. Caso não seja tratado, a extração desse link pode levar à uma coleta repetida devido à barra adicional ao fim da URL devido à utilização da função *Hash* para identificar a referida entrada no *Set* de páginas visitadas.

Com o intuito de identificar o conteúdo das *meta tags* correspondentes para robôs, deparou-se com alguns desafios causados pelos formatos de especificação das diretivas "follow"e "nofollow". Inicialmente, pensava-se que as diretivas eram sempre especificadas em letras minúsculas e separados por vírgulas. Entretanto, verificou-se que a maioria das páginas Web não seguem um padrão para a escrita das *tags*, sendo que alguns escrevem com letras maiúsculas e outros com letras minúsculas, separadas por espaço ou por vírgulas. Deste modo, foi necessário utilizar expressões regulares *case insensitive* para detectar a presença das diretivas.

[1] <www.htmlcleaner.sourceforge.net>

3 Avaliação dos resultados

Após a implementação e validação de seu funcionamento, foram realizados testes de desempenho com o objetivo de relacionar o impacto causado pelo número de *threads* paralelas no desempenho da aplicação

Consoante a isso, os testes foram realizados em máquinas com *hardware* e conexão com a internet distintos, variando o número de *threads* de 10 à 100, com aumento de 20 *threads* entre as etapas do teste. Em cada etapa, foram verificadas quatro execuções e seu valor final foi computado por meio da média aritmética das quatro execuções.

Deste modo, o primeiro teste foi conduzido em um computador com CPU AMD Athlon II X4 645 Processor de quatro núcleos, 8 GB de memória RAM e conexão de 15 Mbps com a *internet*. Os resultados dos testes são mostrados na Figura 5 e na Tabela 1.

Número de threads	**Teste 1**	**Teste 2**	**Teste 3**	**Teste 4**	**Média aritmética**
10	92	107	98	105	100.5
30	58	54	53	48	53.25
50	50	57	54	57	54.5
70	50	46	54	44	48.5
90	51	58	50	59	54.5
100	58	50	56	57	68.25

Tabela 1 – Tempo de execução para o teste 1 em função do número de threads, obtidos em quatro instantes (testes 1,2,3,4) e sua respectiva média aritmética.

Em contraste à outros problemas solucionados com programação paralela, em que não há a necessidade de aguardar entrada e saída, no presente problema é viável utilizar quantidade de *threads* consideravelmente maior do que o número de núcleos do processador. Isso ocorre pois ao realizar uma requisição a thread corrente entra em estado de espera até que seja sinalizado o recebimento da resposta da requisição.

Deste modo, a média aritmética foi computada para cada configuração de execução visando suavizar discrepâncias decorrentes de problemas de rede e escalonamento de CPU. É possível observar que o aumento do número de threads de execução reduziu o tempo de coleta para uma quantidade de *threads* inferior à 70. Por outro lado, valores maiores do que 70 foram prejudiciais e contribuíram para elevação do tempo de execução, devido ao elevado número de trocas de contexto. Portanto, a classe **EscalonadorSimplesTeste** foi implementada de maneira que o número de *threads* é definido de acordo com o número de núcleos presentes no processador da máquina.

Foi realizado um segundo teste em uma máquina com processador AMD Fx 6300

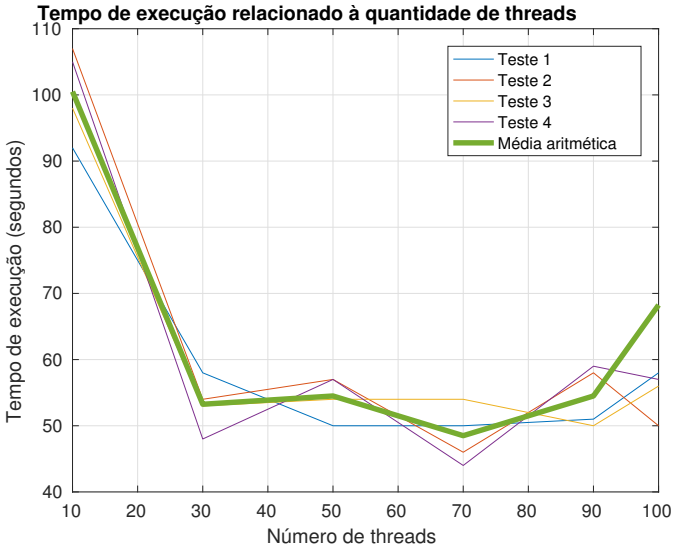

Figura 5 – Gráfico relacionando o tempo computacional e o número de *threads* de execução do teste 1.

Six-Core Processor, 8 GB de memória RAM e conexão média de 5 Mbps. Portanto, percebeu-se a velocidade de conexão com a internet é um fator crítico para a velocidade de coleta. Caso a capacidade de conexão seja saturada pelos **PageFetchers** durante todo o período de execução da aplicação, o aumento do número de *threads* não reduzirá o tempo de execução devido à limitação imposta pela arquitetura de rede.

Número de threads	**Teste 1**	**Teste 2**	**Teste 3**	**Teste 4**	**Média aritmética**
10	107	121	109	137	118.5
30	87	91	102	83	90.75
50	70	84	82	76	78
70	58	80	93	78	77.52
90	82	72	76	74	76
100	62	62	70	76	67.5

Tabela 2 – Tempo de execução para o teste 2 em função do número de threads, obtidos em quatro instantes (testes 1,2,3,4) e sua respectiva média aritmética.

4 Considerações finais

O objetivo deste trabalho foi desenvolver um coletor web de propósito geral de funcionamento paralelo para atender variados cenários de coleta. Este objetivo foi cumprido por meio de pesquisas, análise de dados coletados e revisão de funcionalidades de bibliotecas. Foi estabelecido o requisito de coleta de 500 páginas para este projeto, o mesmo foi cumprido em 44 segundos no melhor caso. Posteriormente foi efetuada a coleta de 40.000 páginas em 1 hora e 1 minuto.

Além disso, foi possível perceber a importância da especificação dos parâmetros da política de exclusão de robôs durante o desenvolvimento de *sites* e hospedagem de servidores.

O presente trabalho contribuiu para o entendimento dos fundamentos dos coletores Web, que devido ao crescente volume de informação distribuída tem se tornado aplicável em diversas áreas de pesquisa e desenvolvimento.

Referências

KUNDER, M. World wide web size. 10 2017. Disponível em: <www.worldwidewebsize.com>.