

Recuperação de Informação - Máquinas de Busca na Web

Arquivos Invertidos

Jordan Silva¹

¹ Instituto de Ciências Exatas

Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

jordan@dcc.ufmg.br

1. Objetivo

Este trabalho tem como objetivo a implementação de um sistema de programas para recuperar eficientemente informação em grandes arquivos armazenados em memória secundária, utilizando um tipo de índice conhecido como arquivo invertido.

A implementação deste sistema foi realizada utilizando a linguagem *C++11*; e como entrada de dados, foi utilizada uma coleção de documentos cedida pelo *LATIN* (Laboratório para Tratamento da Informação)¹.

2. Introdução

O número de documentos² encontrados na internet vêm crescendo rapidamente nos últimos anos, estima-se que existem cerca de 47 bilhões de documentos. Além do grande número de documentos encontrados hoje, eles também se tornaram dinâmicos, assim os conteúdos destes podem sofrer de alterações a cada segundo, aumentando a complexidade no gerenciamento dessas informações. Tendo em vista isto, podemos perceber o grande desafio que existe na tarefa de recuperação de documentos, dado que uma determinada consulta Q de um usuário u , na qual ele expressa a sua intenção por um conjunto de palavras T_q, i e com isso o sistema consiga retornar o melhor conjunto de documentos D_n .

Os sistemas de buscas são os grandes responsáveis pela realização desta tarefa, de uma forma rápida e precisa, de modo a obter informação que é útil ou relevante para o usuário. Essa etapa de encontrar páginas relacionadas com uma consulta é chamada de recuperação de dados, e quando ela é feita em conjunto com alguma heurística que estima a relevância das páginas para os usuários ela é chamada de recuperação de informação. [Baeza-Yates and Ribeiro-Neto 1999]

Nesse trabalho foi construído uma máquina de busca, onde é possível realizar consultas simples. O trabalho teve como objetivo a recuperação de dados, dentro de uma coleção limitada de documentos. Para a recuperação os documentos relacionados à uma consulta de forma eficiente, fez-se necessário a geração de um *índice invertido*, onde cada termo³ encontrado em um documento será representado pela quadrupla $\langle termId, docId, freq, \langle pos \rangle \rangle$, sendo $termId$ é o identificador do termo, $docId$ o identificador do documento, $freq$ a frequência do termo no documento, $\langle pos \rangle$ as posições que o termo aparece no documento [Witten et al. 1999]. Dessa forma, para encontrarmos os documentos relacionados à uma consulta será necessário fazer somente uma interseção dos termos da consulta $T_q i$ com os termos encontrados nos documentos $T_d i$.

¹<http://www.latin.dcc.ufmg.br/>

²Documento é um termo utilizado para páginas de sites na internet.

³Termo é representado por uma palavra do documento.

O trabalho será organizado da seguinte maneira. Na próxima seção será tratada o tema de Recuperação de Informação, e as técnicas utilizadas como base deste trabalho. A Seção 4, será apresentada a arquitetura do projeto e como este foi construído. A Seção 5, ensinará os procedimentos para a execução do sistema. A Seção 6, será abordada as características da coleção de dados utilizada como também a análise e os resultados obtidos. Ao final será apresentada a conclusão do trabalho.

3. Recuperação de Informação

De acordo com [Singhal 2001], a maioria dos Sistemas de Recuperação de Informação em funcionamento hoje, são baseados em uma estrutura de dados conhecida como lista invertida. Essa abordagem permite o acesso rápido a uma lista de documentos que contém um termo, juntamente com outra informação (e.g., o peso do termo em cada documento, as posições relativas do termo em cada documento, etc.). Uma lista típica invertida pode ser armazenada como:

$$t_i \rightarrow \langle d_a, \dots \rangle, \langle d_b, \dots \rangle, \dots, \langle d_n, \dots \rangle \quad (1)$$

que descreve que o termo- i está contido em d_a, d_b, \dots, d_n , e pode armazenar outras informações. As listas invertidas exploram o fato de que dada consulta do usuário, a maioria dos sistemas de recuperação de informação estão interessados em retornar um pequeno número de documentos que contém algum termo da consulta, eliminando assim os documentos que não possuem nenhum termo, através de uma classificação ou pontuação dada para cada termo-documento. [Singhal 2001] também comenta que grande parte desses sistemas mantém essas pontuações dos documentos em um *heap* e no final do processamento ele retorna os *top* documentos mais bem pontuados para a consulta. Uma vez que todos os documentos são indexados pelos termos que contém, o processo de geração, construção, e armazenamento das representações desses documentos é chamado de indexação, e os arquivos invertidos resultantes desse processo são chamados de índice invertido.

4. Arquitetura do Projeto

O projeto foi desenvolvido para realizar a indexação e criação do índice invertido, tal como, possibilitar a recuperação de documentos através de *consultas booleanas* - i.e a concatenação de termos da consulta por meio de **AND** ou **OR**. O projeto foi distribuído em 5 diretórios de forma a agrupar as classes e funções por responsabilidades em comum. Eles foram divididos da seguinte maneira:

- index - Responsável pela leitura e indexação dos documentos;
- libs - Bibliotecas externas para apoio na execução do projeto;
- parser - Tratamento dos termos do documento, auxilia na criação do índice;
- search - Responsável pela execução das consultas através do índice invertido;
- util - Responsável por funções genéricas (e.g., ordenação externa, escrita e leitura em arquivos).

estes diretórios e seu conjunto de classes serão detalhados a seguir.

4.1. Diretório *index*

O conjunto de classes contidas nesse diretório, é responsável pela criação do índice invertido. As classes pertencentes a esse diretório são: *Term*, *Dictionary*, *IndexDocument*, *Indexer*, *IndexTerm*, que serão detalhadas ao longo desse subtópico.

4.1.1. Term

Esta classe é utilizada para armazenamento dos termos do índice em uma memória principal. Ela contém apenas três atributos: um identificador para o termo; o termo em si; e o *seek* (posição do termo) no índice invertido. A classe *Term* conta com alguns construtores para facilitar o instanciamento da mesma.

Term
+ id: unsigned int + indexSeek: unsigned int + term: string
+ Term(unsigned int _id) + Term(unsigned int _id, unsigned int _indexSeek) + Term(unsigned int _id, string _term) + Term(unsigned int _id, string _term, unsigned int _indexSeek) ~ Term()

4.1.2. Dictionary

A classe *Dictionary* é responsável por armazenar o vocabulário em memória principal. O vocabulário é armazenado em uma lista desordenada, contendo como chave a *string* do termo, e uma classe *Termo*4.1.1 para obter mais informações, como a posição de início do termo no índice invertido. Esta classe possui dois métodos *AddTerm* que permitem adicionar os termos ao vocabulário, como também o método *getTerms* que serve para recuperar todo o vocabulário armazenado.

Dictionary
+ terms: unordered_map<string, Term>*
+ Dictionary() ~ Dictionary() + AddTerm(string term): unsigned int + AddTerm(string term, unsigned int seek): unsigned int

4.1.3. IndexDocument

A classe *IndexDocument* é responsável por receber um arquivo *Document* da biblioteca *htmlcxx*4.2, e extrair informações de *url*, *título* e *texto* do documento. Para o funcionamento dessa classe, basta instanciar ela passando como parâmetro um *Document*. Os métodos privados são descritos à seguir:

- *IsScript(string text)* - Verifica se o texto informado é um javascript, se sim, será ignorado como conteúdo do documento;
- *RemoveHeader(string& html)* - Recebe o conteúdo do documento como parâmetro e remove o cabeçalho HTTP.
- *ReadDocument(Document document)* - Recebe um documento como parâmetro (esse método é chamado através do construtor da classe), e realiza a remoção do cabeçalho através do método *RemoveHeader*, logo a seguir, ele varre todo o documento coletando os textos e armazena essas informações em uma variável *text*.

IndexDocument
+ url: string + title: string + text: string
+ IndexDocument(Document document) ~ IndexDocument - IsScript(string item): bool - ReadDocument(Document document): void - RemoveHeader(string& html): bool

4.1.4. Indexer

O papel da classe *Indexer* é de extrema importância para o funcionamento do programa. Esta é a classe responsável por realizar quase todo o processo de indexação. Essa classe que dará início ao processo de indexação, e realizará o controle e chamada das outras classes descritas acima. O construtor da classe *Indexer* recebe como parâmetro três variáveis, o *directory* onde é indicado o diretório que está localizada a coleção de documentos; *mapfile* é a variável contendo o nome do arquivo de índice da coleção a ser indexada; *output* diretório onde será criado o índice invertido e vocabulário.

A classe possui os métodos descritos abaixo:

- *execute(string directory, string mapfile)* - Esse método realiza uma iteração através de todos os documentos da coleção informada, instanciando a classe *IndexDocument* 4.1.3 com cada documento da coleção. Após esse passo, ele possuirá as informações discriminadas de *url*, *título* e *conteúdo* daquele documento, realizando uma chamada para o método *add*, informando esse novo objeto criado e o seu *id* da iteração;
- *add(IndexDocument& document, int documentId)* - O método *add* executará o analisador de conteúdo do documento, com o objetivo de encontrar os termos contidos no documento, removendo espaços, quebras de linhas, remoção de acentos, entre outros tratamentos. Após esse processo, ele irá adicionar os novos termos não vistos ao vocabulário *mDictionary*; e irá começar o processo de gravação da lista invertida, como informado na Equação (1)).
- *dumpVocabulary()* - Gera um arquivo contendo o vocabulário armazenado na variável *mDictionary*;
- *getDictionary()* - Retornará a variável *mDictionary* 4.1.2 que contém o vocabulário da coleção em memória.

Indexer
- mOutputDirectory: string - mWriter: WriterHelper* - mParser: TextParser - mDictionary: Dictionary
+ Indexer(string directory, string mapfile, string output) ~ Indexer() + getDictionary(): Dictionary - execute(string directory, string mapfile): void - add(IndexDocument &document, int documentId): void - dumpVocabulary(): void

4.1.5. IndexTerm

Esta classe auxilia na criação da *lista invertida*, como também na ordenação externa para a criação do *índice invertido*. A classe armazenando informações básicas do *termo-documento*, como o *identificador do termo*, *identificador do documento*, *frequência do termo no documento*, *vetor de posições relativas do termo dentro do documento*. O *IndexTerm* possui os seguintes métodos:

- *print* - Este método retorna o conteúdo da classe formatado;
- *size* - Realiza o cálculo e retorna do tamanho do objeto em bytes;
- *operator>* e *operator<* - Realiza uma comparação entre duas classes *IndexTerm* e informa se a primeira é maior ou menor que a segunda classe.

IndexTerm
+ run: int + termId: unsigned int + documentId: int + frequency: int + positions: vector<int>
+ IndexTerm(unsigned int termId_, int documentId_, int termFrequency_, vector<int> positions_) + print(): string + size(): unsigned int + operator<(const IndexTerm &x, const IndexTerm &y): bool + operator>(const IndexTerm &x, const IndexTerm &y): bool

4.2. Diretório *libs*

Este diretório contém bibliotecas para auxiliar a execução do programa. Elas são:

- *htmlcxx* - Esta biblioteca é utilizada para analisar o conteúdo do documento HTML. Essa biblioteca realiza a construção de uma árvore para representar o documento HTML. Ela será utilizada para a auxiliar a leitura dos documentos da coleção;
- *reader* - Esta biblioteca foi recomendada pela documentação do trabalho, serve para realizar a leitura da coleção à ser indexada. Ela possui a função de descomprimir a coleção e realizar a leitura dela;
- *zlib* - Utilizada pela biblioteca de leitura dos documentos;
- *boost* - Biblioteca utilizada para apoio no projeto. Essa biblioteca provê as bibliotecas do C++ revisadas e melhoradas.

4.3. Diretório *parser*

Este diretório é responsável pela realização por analisar o conteúdo do documento e realizar um tratamento neste. Os tratamentos realizados nesse trabalho foram de remoção de espaços; quebras de linhas; remoção de acentos; remoção de caracteres *HTMLCharacter Entity* [DEV.W3.ORG 2015]; conversão do conteúdo para minúsculo; remoção de caracteres não alfa-numéricos.

4.3.1. TextParser

Essa classe possui os seguintes métodos:

- *Process(string text)* - Esse método executa a análise da variável *text*, realizando os tratamentos descritos acima. Após a realização do tratamento ele executará o método *AddTerm* para adicionar os termos à uma lista de termos encontrados no texto informado;
- *AddTerm(string& token, int pos)* - Recebe o termo e a posição deste no documento e adiciona eles no vetor de termos-posição;
- *removeNonAlphanumerics(string& token)* - Faz a remoção dos caracteres não alfa-numéricos do termo;
- *removeAccents(string& token)* - Remove os acentos do termo;

TextParser
+ terms: unordered_map<string, vector<int>>; - rep[92]: string - sub[92]: string
+ TextParser() + Process(string text) - AddTerm(string& token, int pos) - removeNonAlphanumerics(string& token) - removeAccents(string& token)

4.4. Diretório *util*

Esse diretório possui classes genéricas que podem ser utilizadas em diversos programas ou contextos. Nesse diretório estão contido as seguintes classes *SortFile* destinada para a ordenação externa, e a classe *WriterHelper*, que auxilia na leitura e gravação de arquivos.

4.4.1. SortFile

Essa classe é responsável por realizar a ordenação externa da *lista invertida*, que resultará no *índice invertido*. O construtor da classe *SortFile* recebe o diretório onde foi gerada a lista, e o nome do arquivo da lista. Após a inicialização da classe, o construtor fará uma chamada ao método *execute* que será responsável por realizar a divisão da lista invertida, através do método *split*, e após, realizará a ordenação externa e juntará os arquivos da lista invertida, com o método *merge*.

- *execute(string index)* - Esse método é responsável por realizar a divisão da lista invertida em vários arquivos menores, com o tamanho configurado em uma *constante FILE_SIZE*, atualmente configurada para dividir o índice em vários arquivos de 10MB; também realiza a chamada do método *merge* que é responsável por ordenar e mesclar os arquivos gerados em um arquivo único contendo o *índice invertido*;
- *split(string filename)* - Este método recebe o caminho da *lista invertida* e realiza a divisão dela em vários arquivos semi-ordenados. Esses arquivos têm o tamanho configurado na *constante FILE_SIZE*. A ordenação inicial é realizada dentro de um vetor de *IndexTerm*, preenchido através da lista invertida. Assim que o vetor atingir o tamanho configurado no *FILE_SIZE*, será criado um arquivo de índice novo através do método *createNewIndexFile*, e o vetor será esvaziado e escrito no arquivo, com o método *dumpVector*;
- *createNewIndexFile(string filename)* - Recebe como parâmetro o nome do arquivo à ser criado, e o adiciona em um vetor com o nome dos arquivos criados;
- *dumpVector(vector<IndexTerm>* vectorTerms)* - Recebe um vetor para ser gravado em um arquivo. O vetor será ordenado, gravado em um arquivo, e será esvaziado após;
- *merge()* - Esse método é responsável pela ordenação das *listas invertidas*, o método é executado de forma recursiva, até que a lista de arquivos gerados pelo método *split* esteja vazia. O método lê uma quantidade de arquivos simultaneamente, chamados de *RUN*, retirando assim o primeiro registro de cada *RUN*, logo após o segundo registro e assim sucessivamente. Esses registros são armazenados em uma estrutura *HEAP*, onde é realizada uma ordenação automaticamente. Assim que a estrutura *HEAP* alcançar o tamanho especificado pela *constante HEAP_SIZE*, será retirado o primeiro registro ordenado e gravado em um arquivo de índice. A partir desse momento, a ordem de leitura das *RUNs* será alterada para que seja o próximo registro que a ser incluído no *HEAP* seja da mesma *RUN* que o último registro removido do *HEAP*;
- *write(bool isLastMerge)* - Método responsável por remover do *HEAP* o primeiro registro ordenado e realizar a escrita desse em um arquivo. Este método possui um parâmetro *isLastMerge*, que informa se é a última realização de junção dos índices, caso seja positivo, o método criará um arquivo contendo o vocabulário de *term-id* e a sua posição inicial dentro do arquivo de índice, chamado *seek*, este auxiliará nas consultas ao índice;
- *writeVocabulary(unsigned int id, unsigned int seek)* - Método responsável por escrever no arquivo de vocabulário, o identificador do termo e sua posição no arquivo de índice;
- *mergeVocabulary(string file, string fileSeek, string outputDirectory)* - Recebe como parâmetro o arquivo de vocabulário contendo o identificador do termo e a sua descrição, e um outro arquivo contendo o identificador do termo e a sua posição no *índice invertido*. Esse método realiza uma interseção dessas duas listas, gerando assim um arquivo único de vocabulário contendo o termo, identificador e posição no *índice invertido*.

SortFile
- mOutputDirectory: string - mWriter: WriterHelper* - mVocabularyWriter: ofstream - mQueue: priority_queue<IndexTerm, vector<IndexTerm> , greater<IndexTerm> >* - mQueueFiles: vector<string> - mLastTermIdSeek: unsigned int - mQueueSize: unsigned int
+ SortFile(string directory, string index) + mergeVocabulary(string file, string fileSeek, string outputDirectory): void - createNewIndexFile(string filename): void - execute(string index): void - split(string filename): void - merge(): void - write(bool isLastMerge): int - dumpVector(vector<IndexTerm>* vectorTerms): void - openVocabulary(): void - closeVocabulary(): void - writeVocabulary(unsigned int id, unsigned int seek): void

4.4.2. WriterHelper

Essa classe é responsável por leitura e escrita de arquivos binários. Ela possui os seguintes métodos:

- *WriterHelper(string name, bool writeFile)* - O construtor recebe como parâmetro o nome do arquivo e se ele será aberto para escrita ou somente leitura, através dos parâmetros *name* e *writeFile*;
- *ReadIndex()* - Realiza uma leitura a partir da posição atual, retornando um *IndexTerm*. Funcionará somente para os arquivos de índices gerados pelo programa, ou por uma estrutura de dados idêntica à classe *IndexTerm*;
- *Read(T* obj)* - Realiza a leitura do arquivo para um objeto *genérico T*;
- *Write(IndexTerm& obj)* - Realiza a escrita de um objeto *IndexTerm* em um arquivo;
- *WriteText(string text)* - Realiza a escrita de uma *string* em um arquivo;
- *HasNext()* - Verifica se existe mais informações no arquivo, caso exista, será retornado *true*, caso o arquivo tenha chegado ao final, será retornado *false*;
- *CurrentPosition()* - Retorna a posição atual *seek*, dentro do arquivo aberto;
- *SetPosition(int position)* - Move o *seek*, para a posição informada pelo parâmetro;
- *Open()* - Esse método é chamado somente pelo construtor da classe, onde tentará abrir o arquivo informado;
- *Close()* - Realiza o fechamento do arquivo;
- *Remove()* - Realiza o fechamento do arquivo e a exclusão do mesmo;
- *CheckFile()* - Verifica se o arquivo existe;
- *isOpen()* - Retorna se o arquivo está aberto;
- *getSize()* - Retorna o tamanho do arquivo em bytes;
- *getFilename()* - Retorna o caminho do arquivo;

WriterHelper
- file: FILE* - filename: string - fileSize: unsigned int
+ WriterHelper(string name, bool writeFile) + ReadIndex(): IndexTerm + Read(T* obj): void + Write(IndexTerm& obj): void + WriteText(string text): void + HasNext(): bool + CurrentPosition(): unsigned int + SetPosition(int position): void + getSize(): unsigned int + Close(): void + Remove(): bool + getFilename(): string + isOpen(): bool - Open(bool writeFile): void - CheckFile(): void

5. Executando o Sistema

A execução da aplicação deverá ser realizada via linha de comando, utilizando o **C++11**. O aplicativo pode ser executado por meio do binário **ir** ou compilado através das instruções do arquivo **README** que acompanham o código fonte do programa. O programa possui dois métodos iniciais:

index <directory> <index> [output] - Este método realiza a indexação dos documentos. Os parâmetros *directory* e *index* são obrigatórios, onde respectivamente representam o diretório onde o índice se encontra, e o nome do arquivo de índice. O parâmetro *output* será a pasta que o índice invertido será gerado. Caso o *output* não seja informado, o índice será gerado em uma pasta chamada *output* dentro da pasta onde o programa se encontra.

search [directory] [inverted-index] [vocabulary] [documents] - Este método inicializa o buscador. Neste método, poderá ser informado o diretório(*directory*) onde os estão localizado os arquivos de índice, vocabulário e documentos gerado pelo método de indexação. Todos os parâmetros são opcionais, assim, se os parâmetros não forem informados, os valores assumidos serão: *output* para o diretório; *inverted.index* para o índice invertido; *vocabulary.terms* para os termos do vocabulário; e *file.documents* para o arquivo contendo os documentos da coleção.

- O método **search** possibilitará a realização de recuperação de documentos na coleção. O método pedirá para que seja realizada uma consulta usando uma estrutura *booleana*. Dado a seguinte consulta "**Jordan AND Silva OR UFMG**", o buscador realizará uma consulta retornando os documentos que possuem o conjunto de termos **Jordan Silva** e documentos que possuam o termo **UFMG**.

Descrição	Valor
Tamanho da coleção	4,0GB
Quantidade de documentos	945.642 (100%)
Quantidade de documentos removidos	1.788 (0,18%)
Quantidade de documentos indexados	943.854 (99,82%)
Quantidade de termos indexados	5.213.979
Tamanho do arquivo de índice	3,8GB
Tamanho do arquivo de vocabulário	114,9MB

Tabela 1. Informações da coleção utilizada

6. Análise e Resultados

Nesta seção será analisado a implementação realizada, a coleção indexada, como também os resultados obtidos nesse trabalho. Os testes e a implementação foram realizadas utilizando uma coleção cedida pelo Laboratório para Tratamento da Informação da Universidade Federal de Minas Gerais. A coleção utilizada possui: 945.642 documentos; contendo o tamanho de 4,7GB. Os experimentos foram realizados utilizando um **DELL XPS, Mid 2012; Intel Core i7-2630QM CPU @ 2.00GHz, L1d cache 32K, L1i cache 32K, L2 cache 256K, L3 cache 6144K; 6 GB 1333 MHz DDR3; Ubuntu 14.04.1 LTS.**

A indexação dos documentos foi realizada ignorando documentos que possuem endereços terminados em **.pdf, .doc, .xls, .swf**. Desta maneira, qualquer documento encontrado na coleção que terminar com essas extensões, automaticamente seriam descartados. Como podemos analisar na Tabela 1, com apenas esse tratamento foram removidos 1.788 documentos do processo de indexação, representando uma diminuição de 0,18% da coleção inicial. Esses documentos foram removidos devido a limitações do analisador de conteúdos (htmlcxx4.2) utilizado nesse trabalho.

A Tabela 1 apresenta a quantidade de termos indexados a partir das configurações apresentadas nesse trabalho. Realizamos alguns testes no momento da análise dos termos, como o índice contemplando números, e removendo termos numéricos. Somente com essa alteração obtivemos uma variação de 200MB do tamanho na lista invertida, e 35MB no tamanho do vocabulário, como foi mostrado na Tabela 2.

Descrição	Tempo de Execução	Quantidade de Termos	Tamanho do Índice	Tamanho do Vocabulário
Indexação utilizada	1716 segundos	4.751.387	3,9GB	146,4MB
Indexação com numerais	1763 segundos	6.287.294	4GB	207,4MB

Tabela 2. Comparativo da criação da lista invertida

Após a criação da lista invertida, o sistema realiza a ordenação externa dessa lista. A ordenação ocorre em três que são: (1) A divisão da lista invertida em vários arquivos de **10MB**; (2) a ordenação externa desses arquivos, realizando uma junção de no máximo **50 runs** por vez, utilizando uma estrutura *heap* de **10MB** para ordenação desses registros; (3) e por fim a geração do vocabulário contendo a posição inicial de cada termo no índice ordenado. Esse algoritmo é similar ao algoritmo *External Merge Sort*, assim a sua complexidade no pior caso se dá por $\log_r n/m$, onde r é a quantidade das *runs*; n

a quantidade total de arquivos à serem ordenados; e m é a quantidade de registros que podem ser ordenados na memória[Ziviani et al. 2004].

Esse processo foi realizado com as configurações descritas na Tabela 3, onde foram gerados 385 arquivos, e demorando 631 segundos para ordenar uma lista de 3,9GB (Tabela 4).

Descrição	Valor
Quantidade de RUNs	50
Tamanho do HEAP	10MB
Tamanho dos Arquivos	10MB

Tabela 3. Informações da coleção utilizada

Descrição	Tempo de Execução	Quantidade de Termos	Quantidade de Arquivos
Indexação utilizada	631 segundos	4.751.387	385
Indexação com numerais	765 segundos	6.287.294	405

Tabela 4. Comparativo da ordenação do índice invertido

Durante os testes conseguimos verificar o comportamento do vocabulário, assim como seu crescimento durante a indexação dos documentos. A Figura 6 mostra o crescimento do vocabulário a cada 10 mil arquivos indexados. Observamos assim que o vocabulário tem um crescimento acentuado no início do processo, tendo um ganho de termos significativo nos 70 mil primeiros documentos. Após isso, o crescimento do vocabulário mantém um padrão de 40 mil novos termos a cada 10 mil documentos, tendo uma média de 4 novos termos por documento indexado.

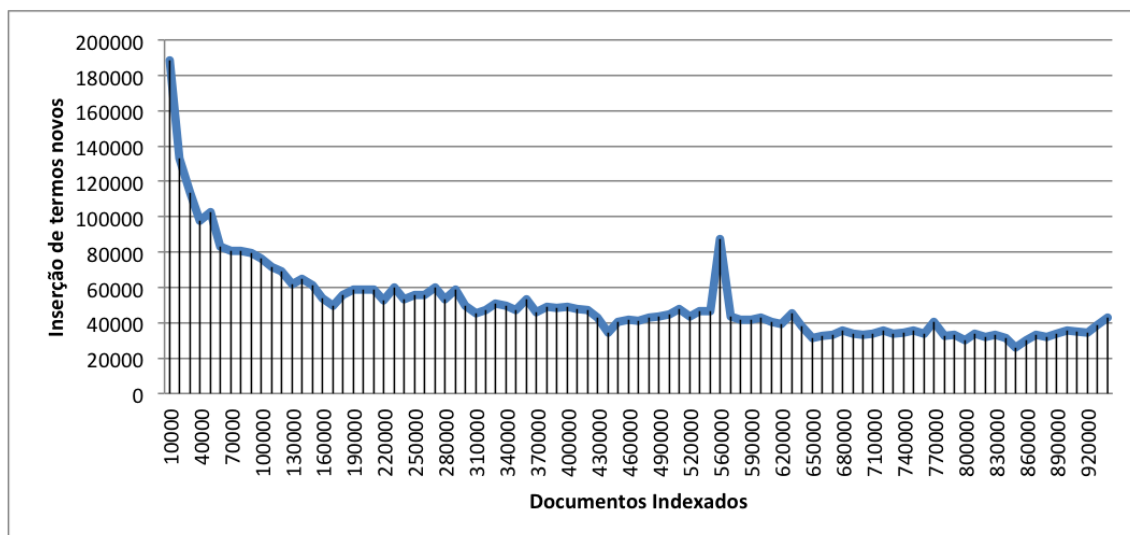
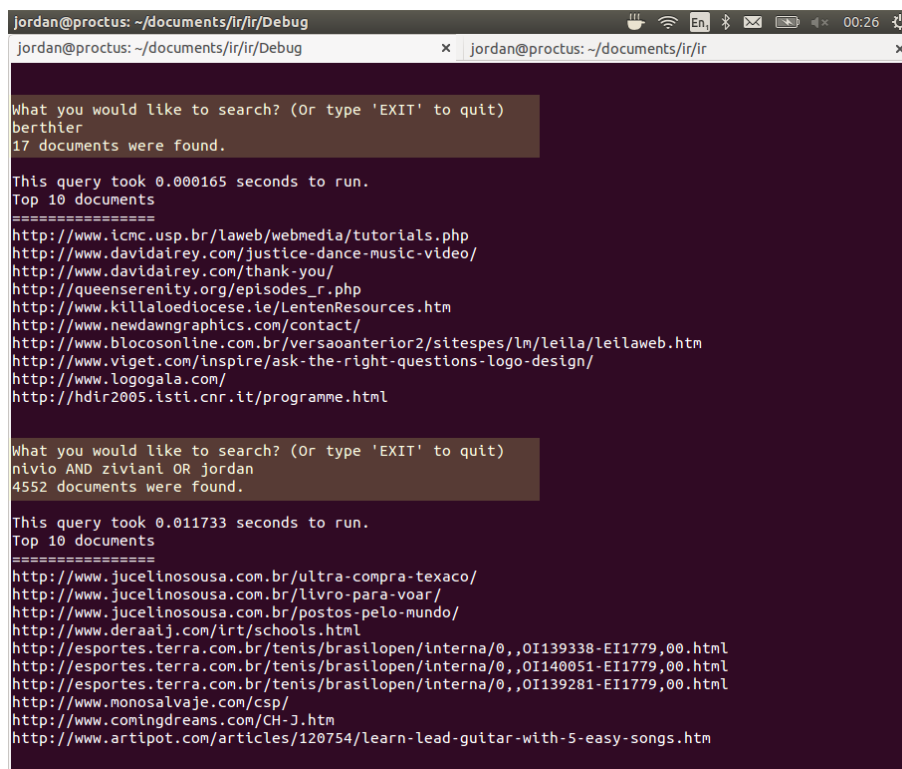


Figura 1. Gráfico de Crescimento do Vocabulário

Após todos os processos: indexação4.1.4, ordenação4.4.1, o programa possibilitou uma recuperação de documentos através de consultas booleanas, como apresentado

na Figura (6). Todas as consultas executadas demoraram menos de 0.01 segundos para serem executadas.



```
jordan@proctus: ~/documents/ir/ir/Debug
jordan@proctus: ~/documents/ir/ir/Debug x jordan@proctus: ~/documents/ir/ir

What you would like to search? (Or type 'EXIT' to quit)
berthier
17 documents were found.

This query took 0.000165 seconds to run.
Top 10 documents
=====
http://www.icmc.usp.br/laweb/webmedia/tutorials.php
http://www.davidairey.com/justice-dance-music-video/
http://www.davidairey.com/thank-you/
http://queenserenity.org/episodes_r.php
http://www.killaloedlocese.ie/LentenResources.htm
http://www.newdawngraphics.com/contact/
http://www.blocosonline.com.br/versaoanterior2/sitespes/lm/leila/leilaweb.htm
http://www.viget.com/inspire/ask-the-right-questions-logo-design/
http://www.logogala.com/
http://hdir2005.isti.cnr.it/programme.html

What you would like to search? (Or type 'EXIT' to quit)
nivio AND ziviani OR jordan
4552 documents were found.

This query took 0.011733 seconds to run.
Top 10 documents
=====
http://www.jucelinosousa.com.br/ultra-compra-texaco/
http://www.jucelinosousa.com.br/livro-para-voar/
http://www.jucelinosousa.com.br/postos-pelo-mundo/
http://www.deraaij.com/irt/schools.html
http://esportes.terra.com.br/tenis/brasilopen/interna/0,,0I139338-EI1779,00.html
http://esportes.terra.com.br/tenis/brasilopen/interna/0,,0I140051-EI1779,00.html
http://esportes.terra.com.br/tenis/brasilopen/interna/0,,0I139281-EI1779,00.html
http://www.monosalvaje.com/csp/
http://www.comingdreams.com/CH-J.htm
http://www.artipot.com/articles/120754/learn-lead-guitar-with-5-easy-songs.htm
```

Figura 2. Realização de buscas no programa

7. Conclusão

Com esta pequena implementação e testes, pude notar a imensidão e as infinitas possibilidades que podem ser exploradas para a melhoria da indexação dos documentos. É um grande desafio desse trabalho, a leitura e tratamento de documentos não estruturados, tentando melhorar a identificação dos termos e construção do índice. Existem várias melhorias a serem realizadas nesse trabalho, algumas dessas são: A melhoria na identificação dos termos para criação do índice; Validação das diferentes codificações de HTML; Reduzir a quantidade de arquivos excluídos da indexação; e alterações da forma de consulta de *booleana* para *vetorial*.

Os objetivos propostos nesse trabalho foram atingidos, realizando assim uma indexação da coleção fornecida, tal como a recuperação dessa informação por meio de consultas *booleanas*. Com ele, foi obtido um grande conhecimento sobre como funciona um sistema de busca, atravessando vários problemas e aplicações para a criação de um sistema simples de recuperação de documentos.

Assim, pretendo em versões futuras, realizar testes mais concretos e ajustes no algoritmo, tal como implementar busca através de outros modelos estudados, realizar a compactação do índice e melhorias no tratamento dos termos dos documentos.

Referências

- Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- DEV.W3.ORG (2015). Character Entity Reference Chart. <http://dev.w3.org/html5/html-author/charref>. [Online; accessed 03-May-2015].
- Singhal, A. (2001). Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43.
- Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann.
- Ziviani, N. et al. (2004). *Projeto de Algoritmos: com Implementações em Pascal e C*, volume 2. Thomson.