Trabalho Prático III - Recuperação de Informação

Prof. Nivio Ziviani e Prof. Berthier Ribeiro-Neto

Evelin Carvalho Freire de Amorim

26 de maio de 2014

Sumário

1 Introdução

Fazer consultar em máquinas de busca é um hábito rotineiro para pessoas no mundo todo. Os resultados das consultas feitas por diversas pessoas são listagens ordenadas de páginas da *web*. A ordenação empregada pela máquina de busca é fundamental para motivar o uso pelas pessoas. Assim muitas pesquisas procuram encontrar uma fórmula eficaz de comparar páginas a fim de determinar qual é a página mais relevante.

As fórmulas desenvolvidas para comparar páginas atribuem uma pontuação para páginas de acordo com o conteúdo dela. Contudo nem sempre apenas o conteúdo consegue representar a relevância em relação a consulta. Por exemplo, consultas que procuram por um *website* são consultas cuja a url do *website* é mais relevante que o conteúdo do *website*.

Uma forma de representar a relevância de uma página, além do conteúdo textual da mesma, é através de referências de outras páginas para ela. A intuição desta ideia é que páginas relevantes são referenciadas por páginas relevantes. Page, Brin, Motwani e Winograd propuseram esta ideia em 1998 e chamaram a estratégia de *Pagerank* [4]. Utilizando fórmulas baseadas em conteúdo e pagerank, Page e Brin criaram uma máquina de busca cujo resultado foi superior aos resultados da época.

Este trabalho procura implementar um sistema que as consultas se baseiam em características textuais e também no cálculo do pagerank proposto originalmente. Este relatório descreve os detalhes da estratégia escolhida e também dos resultados obtidos.

2 Metodologia

O sistema de busca engloba duas partes: o índice e o cálcula da relevância de documentos. O índice, embora já descrito no relatório do trabalho prático 1, foi modificado e será brevemente resumido na Subseção 2.3.

Os *rankings* implementados no trabalho serão brevemente explicados na seção 2.1. A fórmula do *pagerank* será explicada na subseção 2.2.

Além das modificações do índice alguns detalhes de implementação também serão relatados na Seção 2.3.

2.1 Medidas de *Ranking*

Os dois rankings mais populares em recuperação de informação são BM25 e Vetorial. A medida BM25 foi desenvolvida com base em três principios: frequência inversa do documento, frequência do termo e normalização do documento. A Fórmula 1 é a equação do BM25, onde d_j é um documento, q é a consulta, k_i é um termo, N é o número de documentos na coleção e n_i é o número de termos onde k_i ocorre.

$$sim(d_j, q) \sim \sum_{k_i \in q \wedge k_i \in d_j} \log \left(\frac{N - n_i + 0.5}{n_i + 0.5} \right) \tag{1}$$

No entanto como a Fómula 1 possui alguns problemas, como a não normalização do tamanho de documentos. Para tentar resolver este problema a Fórmula 3 foi proposta. Veja que a Fórmula 2, que está contida na formulação do BM25, nada mais é que uma combinação de valores relacionados a frequência do termo.

$$\mathcal{B}_{i,j} = \frac{(K_1 + 1) f_{i,j}}{K_1 \left[(1 - b) + b \frac{len(d_j)}{avg\_doclen} \right] + f_{i,j}} \tag{2}$$

$$sim_{BM25}(d_j, q) \sim \sum_{k_i [q, d_j]} \mathcal{B} \times \log \left(\frac{N - n_i + 0.5}{n_i + 0.5} \right) \tag{3}$$

Mais detalhes sobre o BM25 podem ser vistos em [2].

A fórmula do ranking vetorial faz um produto interno entre o vetor de pesos do documento e da consulta. Cada componente do vetor de pesos do documento é descrito pela Fórmula 5. As componentes do vetor da consulta são similares a do documento, como podemos constatar pela Fórmula 4. Contudo a frequência $f_{i,j}$ do i-ésimo termo na consulta quase sempre é 1 e assim o valor de $\log f_{i,j}$ é 0 e a fórmula para a consulta pode ser representada muitas vezes por apenas por $\log \frac{N}{n_i}$.

$$w_{i,q} = (1 + \log f_{i,q}) \times \log \frac{N}{n_i} \tag{4}$$

$$w_{i,j} = (1 + \log f_{i,j}) \times \log \frac{N}{n_i} \tag{5}$$

Após calcular as componentes do documento e da consulta utilizamos a Fórmula 6 para computarmos a similaridade entre o documento e a consulta.

$$sim(d_j, q) = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^2}} \tag{6}$$

Exemplos com a Fórmula 6 podem ser visto em [2].

2.2 Pagerank

O PageRank foi proposto em 1998 em um relatório técnico [4]. Para compreender o conceito de Pagerank, devemos considerar a web como um grafo cujos nós são as páginas e arestas direcionadas são os links de uma página para outra. A partir desta estrutura assumimos que quando uma página A aponta para uma página B, então é como se A votasse em B. Com esta suposição, consideramos que páginas que são muito votadas são páginas mais "importantes" que páginas pouco votadas.

A ideia do Pagerank é formalizada pela Fórmula 7, onde u é uma página web, F_u é o conjunto de páginas que u aponta, B_u é o conjunto de páginas que apontam para u, c é um fator de normalização e N_u é o número de elementos em F_u.

$$PR(u) = c \sum_{v \in B_u} \frac{PR(v)}{N_v} \tag{7}$$

2.3 Implementação

Considerando os módulos anteriormente implementados não foram modificados, a descrição da implementação foi dividida apenas em dois módulos: Índice e *Ranking*.

2.3.1 Índice

A implementação do sistema de índice está como na arquitetura da Figura . Portanto o funcionamento geral é similar ao do sistema entregue no trabalho prático 1.

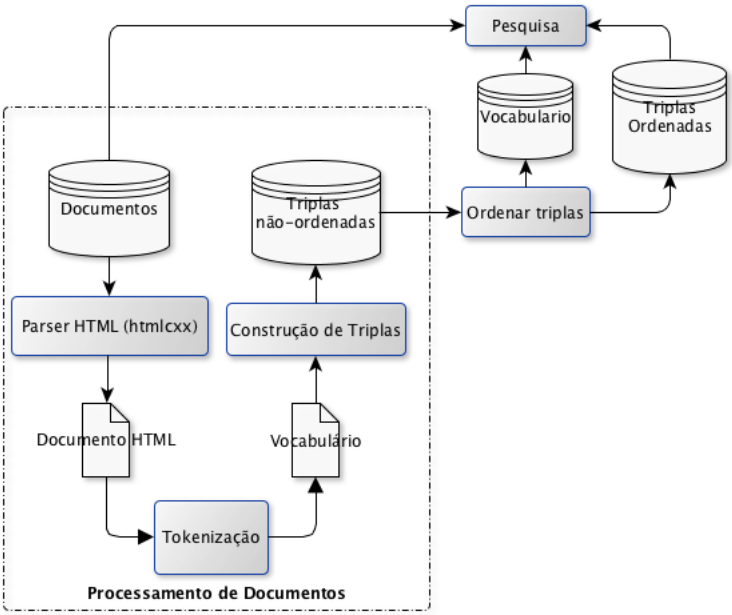

Figura 1: Arquitetura do Índice

Um dos requisitos do trabalho prático atual foi a inclusão de texto âncora de forma a enriquecer o índice. No entanto, por falta de entendimento, no trabalho prático 1 o texto âncora já era indexado. Assim foi feito apenas um teste com um índice sem o vocabulário do texto âncora e com o vocabulário do texto âncora.

Uma melhoria possível no sistema é construir um índice separado para o texto âncora. De acordo com Zaragoza et. al [5], dando um peso diferenciado para título, corpo e texto âncora de uma página, a performance do ranking é superior quando todos estes campos possuem o mesmo peso.

O sistema de indexação também foi modificado para incluir um sistema de indexação de *links*. Esta indexação foi feita utilizando uma tabela *hash* nativa do C++ chamada `unordered_map`[1]. Esta biblioteca foi escolhida em razão da sua rapidez no tempo de resposta na busca. De acordo com a referência gasta em média $O(1)$ para buscar na tabela um elemento e no pior caso $O(n)$.

Nesta tabela de links foi feita a reserva de memória para cada link conforme necessidade. A tabela também é construída apenas com links da própria base, se houver links apontados por páginas da base, mas cujos documentos não existem na base, eles serão apenas ignorados nesta estratégia. O preenchimento desta tabela é feita na classe `Colecao`, na função `ler_arvore_dom`. A seguir o trecho de código no arquivo colecao.cpp em que a tabela de links é preenchida.

```
//verificar se estamos dentro de uma tag de link
if (tag == ‘‘a’’ || tag == ‘‘A’’){
```

```
//esta variavel indica inicio de um trecho do html em que estamos dentro de um link
   islink = true;
   it->parseAttributes();
   string link_href;
   //MAIOR_LINK eh uma constante inicializada com 800
   link_href.reserve(MAIOR_LINK+2);
   link_href = curl.CNormalize(it->attribute("href").second);
   if (link_href.size(>0){
      //copiando os 800 primeiros caracteres
      memset(link_tmp,'\0',MAIOR_LINK+2);
      snprintf(link_tmp,MAIOR_LINK,"%.800s",link_href.c_str();
      //esse aqui vai ajudar a computar o conjunto Bu
      if (indice_links.find(link_tmp) != indice_links.end()){
         indice_links[link_tmp].push_back(idArvore);
       }
      Fu[idArvore-1] = Fu[idArvore-1] + 1;
    }
}else islink = false;
```

Para excluir o texto âncora no vocabulário basta colocar a variável `islink` negada na condição que percorre o vocabulário.

É claro que nesta nova formulação uma maior quantidade de memória é utilizada, visto que incluimos as seguintes estruturas para computar o *pagerank*.

- `indice_link`: Para cada documento da coleção guarda o link e uma lista de inteiros que indica qual o id do documento que aponta para o documento corrente. Assim a memória gasta por esta estrutura será de $\#documentos \times \#outlinks$. Considerando uma média de 20 *inlinks* por documento e que existem 945642 documentos, então teremos (4 bytes $\times 945642 \times 20$) aproximadamente 72 MB em memória das listas de interiros. No entanto devemos contabilizar também as strings que representam as URLs e que a tabela armazena. A Figura 2 exibe a distribuição do tamanho das URLs. No gráfico de distribuição de tamanho de links podemos notar que a maioria das URLs possui até 150 caracteres, então podemos fazer a estimativa de memória neste caso como $\#documentos \times \#caracteresurl$. Computando de acordo com a fórmula da estimativa apresentada temos que $(945642 * 150)$ aproximadamente 135Mb de memória serão consumidos. Portanto, no total esta estrutura consumirá um pouco mais de 207Mb de memória.

- `Fu`: Armazena a quantidade de *links* que a página corrente aponta. Para esta estrutura temos um vetor de 945642 inteiros, portanto a memória consumida será de aproximadamente (945642×4) 3.6MB.

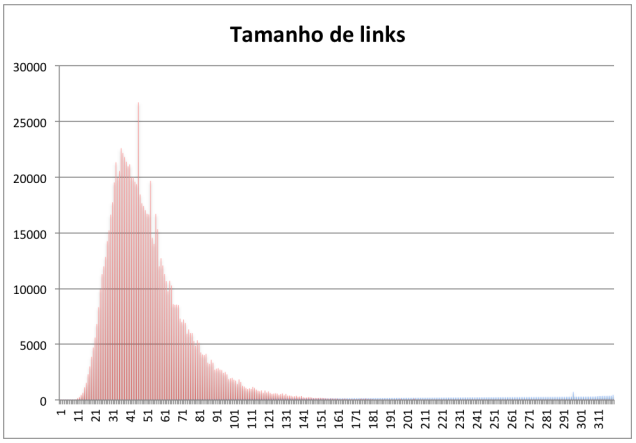

Figura 2: Distribuição do Tamanho dos links

A computação do *pagerank* é feita pela função `computa_info_links`, a qual se encontra no arquivo `colecao.cpp` e na classe `Colecao`. No código a seguir existem dois laços principais, o primeiro computa

uma matriz de *inlinks* de cada página e o segundo computa o *pagerank* propriamente dito. Vamos analisar a memória gasta nesta função de acordo com as estruturas principais:

- `back_links`: Esta estrutura mapeia os *outlinks* de `indice_link` para identificadores inteiros que representam os links. Assim agora não temos mais o mapeamento link -> outlinks, mas sim identificador -> link. Esta estrutura consome aproximadamente ($945642 \times 20 \times 4bytes$) 72MB de memória.
- `pr`: Esta estrutura armazena o *pagerank* das páginas da coleção. Considerando que um `double` tem tamanho de 8 bytes, então esta estrutura consumirá aproximadamente 7.2 MB de memória. A estrutura `old_pr` consome a mesma quantidade de memória.

```
vector<double> Colecao::computa_info_links(string dirEntrada,string nomeIndice){
    //computa o pr das paginas
    ifstream arquivo_link(dirEntrada+nomeIndice,ios::in);
    vector<vector<int> > back_links;
    char* link_tmp = new char [MAIOR_LINK+2];
    vector<double> pr;

    if (arquivo_link.is_open()){
        //inicia matriz de backlinks
        while(!arquivo_link.eof()){
            //...
        }

        vector<double> old_pr(pr);

        int ii = 0;
        //a partir dos backlinks calcular o pagerank
        while(ii< ITER_PR){//calcular x vezes para cad a pagina
            int jj = 0;
            //iterando sobre cada pagina
            while (jj<back_links.size()){
                vector<int>::iterator it_bl_ii = back_links[jj].begin();
                vector<int>::iterator it_bl_ii_fim = back_links[jj].end();
                //iterando sobre o conjunto Bu de cada pagina
                double pr_valor = 0;
                while (it_bl_ii!= it_bl_ii_fim){
                    int k = (*it_bl_ii)-1;
                    if (Fu[k]!=0)
                        pr_valor += old_pr[k]/Fu[k];
                    it_bl_ii++;
                }
                pr_valor = D_FACTOR*pr_valor;
                pr_valor = (1-D_FACTOR) + pr_valor;
                pr[jj] = pr_valor;
                jj++;
            }
            old_pr = pr;
            ii++;
        }

    }else{
        cout<<"Colecao::computa_info_links Problema ao abrir arquivo"<<endl;
    }

    if (link_tmp!=NULL) delete[] link_tmp;
```

```
    return pr;
}
```

Durante a execução de `computa_info_links` o consumo de memória também incluirá as estruturas a seguir:

- `vocabulario` e `vocabulario_invertido`: Aproximadamente estas estruturas consomem $(20 + 4)$ $\times 945642$ bytes, que resulta em 25MB de memória, pois a memória alocada para as palavras em `vocabulario` é a mesma utilizada em `vocabulario_invertido`.

No total o novo índice acrescenta mais 289.8 MB de memória ao gasto original. A complexidade de se calcular o pagerank é realizada após percorrer as páginas, então adiciona-se a complexidade original apenas o tempo que se percorre todos os D documentos duas vezes, i.e., $O(2|D|)$.

A análise feita acima considera apenas a lógica do algoritmo, contudo a aplicação da biblioteca `oneurl` deve também ser levada em consideração. Ao olhar o código fonte da função `CNormalize`(Url.cc) existem várias computações cuja complexidade é $O(|u|)$, onde $|u|$ é o tamanho de uma dada URL. Contudo existem algumas computações que podem utilizar a biblioteca `map` do C++. Assim acredito que no pior caso a complexidade de `CNormalize` é $O(|u|log|u|)$. Supondo que esta computação executa l (número médio de links por página) vezes por documento, nesta fase do sistema a complexidade de tempo de `ler_arvore_dom` muda para $O(|D|(|t||s| + |l||u|log|u|))$ no pior caso, onde $|D|$ é o número de documentos na coleção, na média o tamanho da árvore DOM tem $|t|$ nós e $|s|$ é o número médio de caracteres por documento.

2.3.2 *Ranking*

A computação do *ranking* é feito no arquivo `ranking.cpp`, o qual contém as seguintes classes:

- `Ranking`: Esta classe descreve métodos abstratos e métodos concretos. Os métodos abstratos são métodos que todas as classes de *ranking* devem implementar de acordo com suas características particulares. Os métodos concretos implementados nesta classe são iguais para todas as classes de *rankings* portanto não existe a necessidade de implementação pelas classes filhas.
- `Vetorial`: Esta classe herda a classe `Ranking` e implementa o modelo Vetorial de *ranking*[2].
- `BM25`: Esta classe herda a classe `Ranking` e implementa o modelo BM25 de *ranking*[2]. Nesta fase foi implementada a normalização do BM25, que consiste apenas da divisão da pontuação de cada documento pelo maior valor de pontuação computado.
- `MIX`: Esta classe herda a classe `Ranking` e implementa a *power mean*[3] entre as pontuações do modelo Vetorial e modelo BM25.

Para computar os *rankings* de maneira eficiente, a computação dos pesos de cada documento são pré-computadas e armazenadas no arquivo `wd_compacta.txt`. A criação deste arquivo é controlada pela variável `constroi_wd` no arquivo `pesquisa.cpp`. Por padrão esta variável está com valor `false`. No entanto, na primeira execução do sistema esta variável deve estar inicializada com valor `true` para que este arquivo seja criado. A criação deste arquivo percorre todo arquivo de índice e vai armazenando as frequências dos termos para cada documento. A função que executa esta computação está em `ranking.cpp` e se chama `escreve_wd`. O pseudo-código que computa o peso $wd(i)$ do i-ésimo documento está descrito como a seguir.

3 Resultados e Avaliação

Como já foi mencionado anteriormente a indexação nos trabalhos passados considerou o conteúdo do texto âncora. Contudo para fazer uma comparação mais justa gerei o índice novamente com e sem texto âncora. A Tabela 1 descreve a diferença no tempo da indexação (total) e do tamanho do vocabulário destas duas abordagens.

Abordagem	#triplas	#palavras	Tempo (s)
Sem Texto Âncora	381884617	6154031	9200.66
Com texto Âncora	539785346	8200561	11902.95

Tabela 1: Tabela mostrando o desempenho das Abordagens de Indexação

Como era de se esperar todos os valores de #triplas, de #palavras e de Tempo aumentaram com a inclusão de texto âncora no índice. A explicação óbvia é que o texto âncora possui palavras que antes não eram indexadas e que indexando, geram novas triplas no índice. Mais palavras e triplas também exige mais tempo para ordenação e processamento das mesmas, portanto é natural que o tempo de execução da indexação aumente.

Ambas abordagens da Tabela 1 já incluem a computação do *pagerank*. O cálculo do *pagerank* do meu sistema considera 50 iterações para chegar no valor final. De acordo Page et. al [4] a partir de 50 iterações o *pagerank* converge. O valor da iterações pode ser modificado no arquivo `util.h` através da constante `ITER_PR`. Para computar o *pagerank* também foi considerado o *dumping factor*, que é a probabilidade do usuário, de chegar em uma página u a partir de algum de seus *links* de entrada. O valor desta constante foi inicializado como 0.85. Contudo pode ser modificado através da variável `D_FACTOR`

4 Conclusão

A Manual do Sistema

Três etapas são necessárias para a execução da pesquisa: configuração, compilação e a execução do sistema. As próximas subseções descrevem como fazer estas etapas.

A.1 Configurar

Para a construção do pagerank foi necessárias algumas poucas modificações na construção do índice. Portanto antes de executar a pesquisa é necessária a execução da construção do índice. Estas modificações podem ser relativas a arquivos criados ou podem ser relativos a lógica do algoritmo de indexação.

As modificações implementadas no índice relativas a arquivos são:

1. Criação de um arquivo que armazena informações de cada documento, a saber: tamanho em palavras e pagerank. Este arquivo por padrão se chama info_arquivos.txt;
2. Ao vocabulário foi adicionada a frequência do termo na coleção.

As modificações relativas a implementação estão relacionadas com a construção do pagerank. Como a catalogação de links e posteriormente o cálculo do pagerank. Para a catalogação de links pensei em normalizar as URLs a fim de obter um resultado com mais acurácia. A normalização de links foi feita utilizando a biblioteca oneurl[1]. Logo no pacote deste trabalho existe uma pasta com o código fonte do oneurl. A compilação da biblioteca oneurl exige a instalação do icu4c, cujo download pode ser feito no site `http://site.icu-project.org/download`. No ambiente linux e MacOSX é possível fazer a instalação do icu4c via gerenciador de pacotes. A compilação do oneurl será abordado na próxima seção.

O sistema proposto é compilado através de Makefile. A primeira parte da configuração deve ser feita no Makefile do diretório principal do sistema. No início do Makefile existe um conjunto de variáveis que deve ser modificado conforme os diretórios do usuário. Segue uma lista de tais variáveis e a explicação de cada uma.

- ricode: Diretório onde estão armazenados os códigos objetos da biblioteca CollectionReader;
- urlcode: Diretório com os arquivos de cabeçalho da biblioteca oneurl;
- ridata: Diretório onde se encontra o arquivo com a lista de documentos a serem indexados;
- riindex: nome do arquivo que contém os links dos documentos a serem processados para o índice;

A segunda parte da configuração se encontra dentro no início do arquivo colecao.cpp. A configuração em colecao.cpp engloba os nomes dos arquivos a serem gerados. Seguem as declarações das variáveis como no código fonte.

- `const string Colecao::nome_arquivo_indice="index_compacta.bin";`
- `const string Colecao::nome_arquivo_vocabulario="voc_compacta.txt";`
- `const string Colecao::nome_info_arquivos="info_arquivos.txt";`

[1] `https://github.com/nuoline/oneurl`

A terceira parte da configuração se encontra no início do arquivo pesquisa.cpp e assim como a segunda parte da configuração armazena os nomes dos arquivos a serem gerados ou a serem lidos. Seguem as declarações das variáveis como no código fonte.

- `const string Pesquisa::nome_arquivo_vocabulario = "voc_compacta.txt";`
- `const string Pesquisa::nome_arquivo_indice = "index_compacta.bin";`
- `const string Pesquisa::nome_info_arquivos = "info_arquivos.txt";`
- `const string Pesquisa::nome_dir_saida = "saida/"; //guarda resultados da pesquisa`

A quarta parte da configuração é a escolha do

A.2 Compilar

Caso a configuração tenha sido feita de forma correta a compilação do sistema segue como próximo passo. As subetapas da compilação da compilação são: compilação do oneurl, compilação do índice e compilação da pesquisa.

A.2.1 Oneurl

A compilação do oneurl é feita através da sequência de comandos a seguir:

```
cd oneurl-master/ && make && cd ..
```

Caso aconteça algum problema é possível que esteja relacionada com a biblioteca icu4c, pois o pacote do oneurl abrange a biblioteca icu4c já compilada. Isso não é recomendado visto que a instalação pode ocorrer em uma arquitetura diferente da arquitetura onde a icu4c foi compilada. Nesta situação basta modificar o Makefile da pasta oneurl-master para apontar para o icu4c que foi compilado nativamente em sua máquina.

A.2.2 Índice

A compilação do índice é feita através do comando:

```
make index
```

Assumindo que os código objetos do CollectionReader já existam, caso contrário é necessário executar make ziplib antes de make index.

A.2.3 Pesquisa

A compilação da pesquisa é feita através do comando:

```
make pesquisa
```

A.3 Testar

Referências

[1] C++ reference `http://www.cplusplus.com/reference/unordered_map/unordered_map/`, 2000.

[2] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] David W. Cantrell and Eric W. Weisstein. "power mean." from mathworld–a wolfram web resource. `http://mathworld.wolfram.com/PowerMean.html`.

[4] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[5] Hugo Zaragoza, Nick Craswell, Michael J Taylor, Suchi Saria, and Stephen E Robertson. Microsoft cambridge at trec 13: Web and hard tracks. In *TREC*, volume 4, pages 1–1. Citeseer, 2004.