Algoritmos e Estruturas de Dados 3
Trabalho Prático 0
Similaridade de Textos

André Taiar Marinho Oliveira

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

taiar@dcc.ufmg.br

Agosto de 2010

Resumo. *Recuperação de Informação (RI) é uma área da computação que lida com o armazenamento de documentos e a recuperação automática de informação associada a eles. É uma ciência de pesquisa sobre busca por informações em documentos, busca pelos documentos propriamente ditos, busca por metadados que descrevam documentos e busca em banco de dados, sejam eles relacionais e isolados ou banco de dados interligados em rede de hipermídia, tais como a World Wide Web.*
Uma boa forma de classificar, definir e reconhecer semelhanças entre textos é definindo o seu assunto de acordo com as suas palavras chaves. Com o crescimento da utilização de sistemas de Recuperação de Informação e sua utilização massiva na atualidade, o problema de se descobrir palavras-chave de um texto tem sido muito explorado.

1. Introdução

Neste trabalho foi desenvolvido um sistema que, dado um conjunto de textos qualquer, encontramos para cada elemento o seu tópico/palavras-chaves. De acordo com a ocorrência das palavras-chaves nos textos, podemos definir algumas métricas de semelhança entre os eles e, assim, identificar textos que têm conteúdo semânticamente relacionado de uma forma automática.

Esse tipo de análise de conteúdo semânticamente relacionado vem sendo amplamente utilizado para conjuntos cada vez maiores de informação. A Web é um exemplo de rede com conteúdo vasto e que crescimento vertiginoso. Organizar e encontrar toda essa informação de forma relevante através em um conteúdo tão vasto e fragmentado em sites, blogs, wikis, redes sociais, etc, é uma importante aplicação deste trabalho e do que é desenvolvido com relação à Recuperação de Informação.

Aqui foram implementados algoritmos simples operando sobre uma estrutura de dados de pesquisa em memória bem eficiente e dinâmica. Foram avaliadas métricas simples de classificar o conteúdo dos textos por palavras-chaves e, posteriormente calcular o quão relacionados estes textos são. Ao final, farei a análise de quão boa foram as métricas propostas para avaliar a semelhança entre os conteúdos e variar as diversas considerações para obter uma visão mais ampla do funcionamento do sistema.

2. Solução Proposta

A solução proposta para os cálculos sobre as frequencias e ocorrencias das palavras utiliza um conjunto de estruturas de dados que contém informações sobre o termo em si (a palavra especificamente), o texto aonde ela ocorreu e quantas vezes ocorreu naquele texto. Em diferentes partes do algoritmo, esta estrutura está organizada de diferentes formas: às vezes como uma árvore binária, como um vetor de árvores binárias, como uma lista encadeada e como um vetor nos casos em que exigiam ordenação.

Para criar uma estrutura de pesquisa eficiente tanto em termos de espaço quanto custo computacional, optei por armazenar todo o índice de ocorrências dos termos em forma de uma árvore binária. Cada nó dessa árvore tem uma lista encadeada de ocorrências do termo que armazenam em cada célula o texto em que aquele termo apareceu e quantas vezes apareceu nesse texto.

Primeiramete, é criado um índice lendo um por um os arquivos passados como entrada para serem analisados. Cada termo desse arquivo é inserido na estrutura da árvore e servirá como nosso índice/vocabulário.

1: Leitura do índice

```
foreach Documento do
    foreach termo válido do
        if o termo já apareceu em algum texto then
            if o termo já apareceu neste texto then
                incrementa o contador de ocorrências correspondente ao termo no texto;
            end
            else
                insere o texto na lista e contabiliza o contador de ocorrências para este
                termo;
            end
        end
        else
            insere o nó referente ao termo, insere o texto na lista e contabiliza o contador
            de ocorrências para este termo;
        end
    end
end
```

Em uma aplicação real, muitos passos podem ser feitos para determinar efetivamente o que é e o que não é um termo relevante e, dessa forma, poupar processamento e espaço em processos de indexação. Alguns desses passos correspondem à análise de Stop Words [1], remoção de prefixos e sufixos das palavras [2], reconhecimento sintático, entre outras. Em nosso sistema, a única análise feita é quanto ao número de caracteres que a palavra contém. Para uma aplicação padrão da indexação, utilizei o mínimo de 3 caracteres por palavra (parâmetro que será variado nas avaliações experimenais).

Após indexar todos os textos, conseguimos obter vários parâmetros úteis para expressar a relevância das palavras-chave neste contexto. O primeiro aspecto a ser lev-

[1] Stop Words são palavras consideradas sem valor semântico para a análise de tópico de um texto (como artigos e preposições, por xemplo). Referência.

[2] É um processo que simplifica as palavras removendo seu prefixo e sufixo (caso tenha) e dessa forma reconhece o padrão de formação daquela palavra.

ado em consideração é o número de ocorrências de uma palavra. Como sugerido na especificação, assumi que, para um termo ser considerado relevante ele deveria aparecer na minoria dos textos apresentados (aparecer em menos de 50% dos textos) aumentando o índice de discriminação que ele potencialmente tem. Esse parâmetro foi colocado (assim como o tamanho mínimo dos termos) de forma a ser facilmente modificado para obtermos melhores análises experimentais.

Após termos uma referência sobre quais palavras serão potencialmente relevantes para analisarmos as semelhanças entre os textos, podemos reconstruir novamente um índice. Dessa vez, eu analiso cada texto e incorporo ao seu índice apenas as palavras relevantes, colocando cada índice em um vetor que terá o índice de um texto em cada posição.

Com estes elementos calculados podemos partir para a finalização do trabalho. Para retornar as palavras chaves de cada texto, basta percorrer o índice individual de cada texto. Porém, como as palavras chaves precisam ser retornadas devemos primeiramente ordená-las de acordo com as suas ocorrências naquele texto. Para isso, lemos o índice e armazenamos os termos e suas ocorrências um um vetor que será posteriormente ordenado decrescentemente, obtendo dessa forma as palavras-chave de um texto ordenadas por sua relevância.

O segundo item a ser retornado pelo programa é um arquivo contendo a lista dos textos analisados se os texos mais semelhantes à ele. O número de textos a ser retornado é arbitrário. Nas execuções do trabalho, retornamos quantidades de 5 a 8 indicações de textos.

2: Identificação de textos semelhantes

```
foreach Documento do
    PalavrasChave ⟵ recebe as palavras-chave do Documento e o seu número de
    ocorrências naquele texto;
    OrdenaOcorrenciaDecrescente(PalavrasChave);
    Imprime lista de palavras-chave;
    foreach palavras-chave do Documento do
        Ocorrencias ⟵ vetor de textos e numero de ocorrencias da palavra-chave por
        texto;
        foreach Documento em que a palavra-chave ocorreu do
            PotencialDeSemelhanca[DocumentodeOcorrencia]+ ⟵ ocorrencias da
            palavra-chave no Documento de Ocorrencia;
        end
    end
    OrdenaPotenciaDecrescente(PotencialDeSemelhanca);
    Imprime lista de N Documentos mais similares;
end
```

Encerrada a explicação sobre os principais algoritmos e as estruturas de dados utilizados, e como eles se combinaram na solução do problema, segue abaixo uma breve análise sobre a análise de complexidade dos principais algoritmos e mais alguma explicação sobre as estruturas que eventualmente tenha faltado acima.

2.1. Análise da solução

2.1.1. Ordenação

O algoritmo de ordenação utilizado foi o Quicksort que tem ordem de complexidade $n(\log(n))$ para casos médios.

2.1.2. Dicionário

Armazena cada termo, o documento em que ocorreu e quantas vezes ocorreu naquele documento. O Dicionário foi organizado como uma árvore binária de pesquisa em que cada termos e suas ocorrências estão em um nó da árvore. Dessa forma, cada nó ainda conta com 2 ponteiros (um que aponta para a sub-árvore da direita e um que aponta para a sub-árvore da direita).

3: Nó da Árvore
termo;
lista de ocorrencias;

Tabela 1. Análise de Complexidade do Dicionário

Operação	Custo (casos médios)
Inserção na Árvore	$O(\log(n))$
Pesquisa na Árvore	$O(\log(n))$
Caminhamento na Árvore	$O(n)$

2.1.3. Lista de Ocorrências

Precisamos de uma lista de ocorrências para guardar todas as ocorrências que variavam por termo e documentos. Cada nó do dicionário contém uma lista de ocorrências.

4: Item da lista
identificador do documento;
ocorrências no documento;

As operações em lista são sempre $O(n)$ para o pior caso.

3. Código

3.1. Módulos

O programa foi separado em quatro módulos:

- **Vocabulário:** controlou das estruturas responsáveis pelo armazenamento dos termos e o cálculo de toda lógica de funcionamento.
- **Entrada e Saída:** controlou os arquivos que foram lidos e escritos pelo programa bem como a validação da entrada pela linha de comando. Outra função importante deste módulo foi filtrar os termos que não eram considerados palavras relevantes para o funcionamento do programa.

- **Util:** contém algumas funções que não se encaixavam bem nos outros módulos, além de ter o método de ordenação utilizado.
- **Principal:** interagiu com todos os outros módulos retornando erro quando alguma irregularidade era colocada, controlou o fluxo de execução do programa.

3.2. Entrada e saída

Para não haver problemas com a leitura da linha de comando, foi utilizado um utilitário do sistema Linux chamado **getopt** que permite uma leitura mais robusta dos argumentos passados ao programa. O comando para execução do programa definido para a linha de entrada foi:

```
#: ./tp0 -a <A> -b <B> -c <C>
```

Sendo:

- **A** - caminho para um arquivo de entrada que contém uma lista de outros arquivos (um arquivo por linha);

```
texto1.txt
texto2.txt
texto3.txt
texto4.txt
...
textoN.txt
```

- **B** - caminho para o arquivo de saída que terá a lista de arquivos da entrada seguidos por suas respectivas palavras-chave;

```
texto1.txt;palavraA;palavraB;palavraC;...;palavraZ;
texto2.txt;palavraA1;palavraB1;palavraC1;...;palavraZ1;
texto3.txt;palavraA2;palavraB2;palavraC2;...;palavraZ2;
texto4.txt;palavraA3;palavraB3;palavraC3;...;palavraZ3;
...
textoN.txt;palavraAN;palavraBN;palavraCN;...palavraZN;
```

- **C** - caminho para o arquivo de saída que terá a lista de arquivos da entrada seguidos por uma lista de textos semelhantes à ele.

```
texto1.txt;textoA.txt;textoB.txt;...;textoZ.txt;
texto2.txt;textoA1.txt;textoB1.txt;...;textoZ1.txt;
texto3.txt;textoA2.txt;textoB2.txt;...;textoZ2.txt;
texto4.txt;textoA3.txt;textoB3.txt;...;textoZ3.txt;
...
textoN.txt;textoAN.txt;textoBN.txt;textoCN.txt;...;textoZN.txt;
```

Todos os arquivos e/ou palavras chaves que forem listados devem estar na mesma linha iniciada pelo arquivo com o qual se relaciona e devem estar separados entre si apenas por um caractere ”;”.

Como convenção, adotei uma regra que havia sido definida no fórum da disciplina e optei por utilizar textos sem acentuação nos caracteres, sem ”til”e sem cedilha. Para tal, antes de executar o programa eu utilizei o script de minha autoria que foi disponibilizado no fórum para tratar todas as entradas e obtê-las sem acentuação.

3.3. Compilação

O compilador utilizado neste trabalho foi o GCC (adotado como padrão para a disciplina) e o comando para compilar o programa através do GCC é:

```
#: gcc -o tp0 main.c io.c io.h vocabulario.c vocabulario.h \\
          util.c util.h
```

caso estejam todos os arquivos dentro do mesmo diretório.

Como pedido na especificação, foi feito um arquivo Makefile com o qual é possível compilar o programa com o comando:

```
#: make build
```

E podemos compilar e executar o programa com o comando:

```
#: make run
```

que fará, além da compilação, com que o programa rode e execute a sua rotina para uma entrada já tratada e sem acentuação, "til"ou cedil hasdisponibilizada juntamente com o diretório do trabalho.

4. Avaliação Experimental

O que pudemos avaliar experimentalmente com a execução do programa é que apesar de ter encontrado bem as palavras-chave de cada texto (os termos considerados foram realmente relevantes), o algoritmo para clusterizar os textos não foi tão eficiente assim.

Aparentemente, textos com muitas palavras foram considerados muito mais que outros textos menores. Esta falha provavelmente se encontra no algoritmo do cálculo do potencial de semelhança entre os textos, descrito em alto-nível no algoritmo 2. Em uma próxima versão será interessante normalizar matemáticamente essa contagem para evitar esse tipo de discrepância de resultados.

Como não fiz medições do tempo de execução do programa para diversas entradas, a avaliação da velocidade do programa para diferentes entradas será omitida.

5. Conclusão

Neste trabalho construímos um sistema que permite separar por critérios de similaridade (clusterizar) um conjunto de textos arbitrário através da identificação de tópicos/palavras-chave desse texto.

O trabalho conseguiu cumprir o propósito de praticar a linguagem C e o estudo de um problema mais complexo do que o de costume. A utilização massiva de estruturas de pesquisa foi uma novidade. Uma decisão extremamente importante para o trabalho foi a de utilizarmos uma estrutura em forma de árvore binária ao invés de outras que poderíamos utilizar como, por exemplo, tabelas hash o que provavelmente fez o programa executar com uma velocidade maior e gastando uma quantidade menor de espaço em memória.

A análise da variação da qualidade da similaridade avaliada pelo programa é uma avaliação subjetiva pois depende da leitura e interpretação dos textos retornados por parte de uma pessoa (melhor avaliadora de similaridade de conteúdo possível). Porém, ao

analisarmos a ocorrência de palavras chaves vemos que o sistema encontrou palavras que tinham realmente a ver com o conteúdo daquele texto e que não eram tão decorrentes assim nos outros (tinham certo caráter discriminativo entre os textos analisados).

Algumas melhorias que poderiam ser consideradas neste trabalho são:

- Um melhor tratamento dos textos de entrada (eliminando acentuação e outros caracteres inválidos dentro do programa);
- Utilização de Stop Words, retirada de prefixo e sufixo e outros tipos de técnicas aplicadas em recuperação de informação com o objetivo de maximizarmos o aproveitamento dos termos contidos no índice;
- Uma melhor análise de complexidade de espaço e tempo de execução do programa, talvez a utilização de uma outra estrutura de dados para o índice seja mais eficiente em termos de execução;
- Uma modelagem mais complexa da análise de frequência das ocorrências de termos relevantes nos documentos analisados;
- Uma modelagem mais complexa do cálculo de similaridade entre os textos a partir de suas/seus palavras-chave/tópicos.