Trabalho Prático 3:

Máquina de Busca Avançada

Erick Sunclair Santos Batista - 2020026877

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

ericksunclair@ufmg.br

1. Introdução

Um motor de pesquisa é responsável por procurar palavras-chave fornecidas pelo utilizador em uma base de dados. O buscador é composto de três parte: o crawler coleta os documentos, o indexador lê os documentos e constrói o índice invertido o processador de consultas consulta o índice e ordena os documentos de acordo com a relevância com a consulta.

Nesse trabalho foram desenvolvidas as duas últimas partes do buscador, o indexador e o processador de consultas, para buscar palavras especificadas em um banco de documentos e retornar um ranking com os 10 documentos mais relevantes com relação à consulta, sendo que esta consulta leva em conta o Modelo Espaço Vetorial para relacionar e calcular a similaridade entre a consulta e um documento.

A seção 2 aborda as características do código implementado. A seção 3 apresenta uma análise de complexidade de tempo do programa. A sessão 4 detalha as estratégias de robustez utilizadas para tornar o programa mais seguro e menos passível a erros. A sessão 5 possui a descrição dos testes realizados e as saídas obtidas para esses testes. A sessão 6 apresenta os resultados dos testes realizados e uma análise desses resultados obtidos. Por fim, a seção 7 possui a conclusão do trabalho desenvolvido.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection e os testes foram realizados em um computador com Windows 10, intel core i5 3.4 GHz e 8GB de memória RAM.

O sistema possui duas constantes globais: num_max_palavras, que representa o tamanho máximo de vocabulário que o programa consegue armazenar (levando em conta que repetições de palavras não contam a mais no vocabulário, sendo apenas uma palavra) e num_max_palavras_consulta, que representa o máximo de palavras que o documento de consulta terá. Essas variáveis podem ser alteradas de acordo com a necessidade do usuário, tamanho do banco de dados e características das consultas a serem feitas.

O programa inicialmente acessa o arquivo de stopwords (palavras que não possuem muito significado e utilidade e, portanto, não são consideradas na consulta) e cria um vetor com esses termos. Em seguida são tratados os documentos do corpus, ou seja, são retirados símbolos, palavras com caracteres numéricos e stopwords e todas as palavras são convertidas para lowercase. Cada documento do corpus é tratado e cria-se uma cópia dele já tratado no diretório “corpus_tratado/”.

Em seguida o programa conta as palavras de toda a base de dados (dos documentos já tratados) e o número de documentos. São criadas duas listas: uma com as id's de todos os documentos e outra com todas as palavras do vocabulário (sem contar repetições). O próximo passo é preencher o hash com os termos e suas ocorrências. Para organizar todos esses termos foi usado um Hash como tipo abstrato, Esse Hash consiste em uma lista de linhas (uma tabela), onde cada linha representa um termo e possui o termo em si (string), o valor hash desse termo (double), o número de ocorrências desse termo (em quantos documento ele aparece) e um vetor de ocorrências (sendo cada ocorrência um par de inteiros – id do documento, frequência do termo no documento). O valor hash de cada termo é calculado pela função hash mostrada na Imagem 2.1 abaixo (compute_hash). Essa função calcula um valor hash de uma string de entrada com chance de colisão muito baixa.

```
long long Hash::compute_hash(string const& s)
// Descricao: adiciona uma ocorrencia ao hash
// Entrada: termo - termo do qual é a ocorrencia
// Saida: valor hash do termo
{
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (auto it=s.begin(); it!=s.end(); it++) {
        hash_value = (hash_value + (*it - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Imagem 2.1 – Função Hash Utilizada

Após o preenchimento do Hash com todas as ocorrências de cada palavra, o documento com a consulta a ser realizada é aberto e é calculada a similaridade entre cada documento do corpus e a consulta realizada. Essas similaridades são armazenadas em um vetor de doubles. Em seguida são impressas as id's dos 10 documentos com maior similaridade não nula (se a similaridade é nula o documento não entra no ranking). Se dois documentos possuem mesma similaridade o de menor id é impresso primeiro.

3. Estudo de Complexidade

Nessa seção será analisada a complexidade temporal teórica de cada trecho e função do programa e do sistema como um todo. A etapa de salvamento das stopwords possui complexidade O(n), sendo n o número de stopwords, enquanto o tratamento dos documentos possui complexidade O(n*m*i*(j+k)), sendo n o número de documentos no corpus, m o número de palavras em cada documento, i o tamanho de cada palavra, j o tamanho de um trecho da palavra (pois ela é separa em trechos se houverem espaços dentro dela) e k o número de stopwords.

Para contar as palavras e documentos a complexidade é O(n*m*i), sendo n o número de documentos, m o número de palavras por documento e i o número de palavras encontradas até então. O preenchimento do Hash é O(n*(m*i+i)), Com n, m e i tendo os mesmos significados da última análise. O cálculo da similaridade é O(n+i*(m+n*m)), sendo n o número de palavras da consulta, i o número de documentos do corpus e m o número de palavras do vocabulário. Por fim, a impressão do ranking possui complexidade O(10*n), ou seja O(n), em que n é o número total de documentos do corpus.

O tipo abstrato de dados Hash também possui algumas funções que podem ter sua complexidade analisada. O construtor dessa classe tem complexidade O(n), com n = número de palavras do vocabulário, o número de linhas da tabela Hash. O destrutor tem a mesma complexidade. A função add, reponsável por adicionar uma ocorrência de um termo no Hash, também tem a mesma complexidade O(n). Compute_hash é O(i), sendo i o tamanho da string da qual será calculado o valor hash. Get_termo e get_hash retornam respectivamente um termo ou o hash de um termo e tem complexidade O(1). A função freq._termo_doc retorna quantas vezes um termo aparece em um documento e tem complexidade O(n*i), sendo n o número de termos do Hash e i o número de ocorrências de cada termo. A última função é quantos_doc_tem e retorna o número de documentos que possuem um termo. Sua complexidade é O(n), com n ainda representando o número de termos do Hash.

4. Estratégias de Robustez

Como estratégias para tornar o nosso programa mais robusto e tratar possíveis erros a principal ação tomada foi sempre terminar o programa quando a tentativa de abrir algum arquivo não for bem-sucedido, imprimindo também no terminal uma mensagem informando qual documento teve problema ao ser aberto.

Além disso, o programa também é terminado com uma mensagem caso o número de palavras do corpus exceda o número máximo de palavras ou o número de palavras por consulta exceda o número máximo de palavras por consulta, as duas constantes do nosso programa. Caso algum desses casos ocorra essas constantes podem ser alteradas no cabeçalho do arquivo main.cpp do programa, no entanto estes valores já estão regulados e são suficientes para o corpus small e para o corpus full e para as consultas a serem realizadas. O número máximo de palavras está definido como 27000 (o corpus full chega a aproximadamente 22000) e o número máximo de termos por consulta está definido como 100 (sendo que as consultas normalmente possuem menos de 10 termos).

5. Testes

O teste realizado utilizou um corpus com 201 documentos (diretório “colecao_small/”) e fez 10 consultas para esse corpus, sendo as consultas as seguintes:

- Consulta 1 – “laptop”
- Consulta 2 – “tablet”
- Consulta 3 – “ipad”
- Consulta 4 – “android”
- Consulta 5 – “vídeo game”
- Consulta 6 – “usb cable”
- Consulta 7 – “iphoto”
- Consulta 8 – “iridium”
- Consulta 9 – “card”
- Consulta 10 – “imac apple”

Os resultados obtidos (ranking dos 10 documentos mais similares com a consulta) foram:

- Ranking 1 – “7425 14681 13864 14850 14259 8877 8763 1599 15323 4536”
- Ranking 2 – “8611 14506 6125 11220 11221 239 490 15636 12910 10856”
- Ranking 3 – “9917 3180 3647 13678 3894 5332 5384 6187 17571 12359”

- Ranking 4 – “3629 5174 13682 15636 12804 12509 11002 10002 14128 14843”
- Ranking 5 – “5668 16965 5548 1291 16213 16870 15264 17684 8930 8140”
- Ranking 6 – “17801 16722 16705 649 5101 16672 16661 18547 17593 8082”
- Ranking 7 – “7442”
- Ranking 8 – “16281”
- Ranking 9 – “16528 1253 18111 1499 16360 16361 2353 8764 11335 15933”
- Ranking 10 – “2353 70 321 64 315 4871 1246 5237 4109 16052”

Podemos perceber que no caso das consultas 7 e 8 apenas um documento possuía similaridade com a consulta, por isso apenas um documento aparece no ranking.

6. Análise Experimental

Foi usado o gprof para analisar os tempos de execuções das funções do nosso programa e sua parcela no tempo total e o resultado obtido pode ser observado na Imagem 6.1 abaixo. São mostrados os componentes mais relevantes, sendo que os demais praticamente não influenciam no tempo total. Podemos ver que as funções freq_termo_doc e quantos_docs_tem são responsáveis por grande parte do tempo de execução, seguidos de alguns métodos com strings (alocação e comparação de string) e logo depois da função compute_hash, e, mais abaixo, a função add, com tempos menores, mas consideráveis, mas o maior tempo se deve ao método __enable_if, relacionado à definição de alguns tipos. A execução total das 10 consultas totalizou aproximadamente 3 minutos 20 segundos.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 40.89      1.84     1.84 1281745056     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__type std::
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
 18.89      2.69     0.85   358986     2.37     5.22  Hash::freq_termo_doc(std::__cxx11::basic_string<char, std::char_traits<char
 18.67      3.53     0.84   717972     1.17     2.61  Hash::quantos_docs_tem(std::__cxx11::basic_string<char, std::char_traits<ch
  9.56      3.96     0.43                             std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<cha
  4.89      4.18     0.22 144940013     0.00     0.00  std::char_traits<char>::compare(char const*, char const*, unsigned int)
  1.56      4.25     0.07  2404374     0.03     0.06  Hash::compute_hash(std::__cxx11::basic_string<char, std::char_traits<char>,
  1.33      4.31     0.06                             std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<cha
  0.67      4.34     0.03 16987022     0.00     0.00  bool __gnu_cxx::operator!=<char const*, std::__cxx11::basic_string<char, st
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > const&, __gnu_cxx::__normal_iterator<char const
  0.67      4.37     0.03     4112     7.30    41.34  Hash::add(std::__cxx11::basic_string<char, std::char_traits<char>, std::all
  0.67      4.40     0.03                             _fu7___ZSt4cout
  0.44      4.42     0.02 14582648     0.00     0.00  __gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
  0.44      4.44     0.02                             memcmp
  0.22      4.45     0.01 33974044     0.00     0.00  __gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
  0.22      4.46     0.01 14582648     0.00     0.00  __gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
  0.22      4.47     0.01  1184493     0.01     0.01  Hash::get_termo[abi:cxx11](int const&)
  0.22      4.48     0.01                             std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<cha
  0.22      4.49     0.01                             std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<cha
  0.22      4.50     0.01                             _fu3___ZSt4cout
  0.00      4.50     0.00 14582648     0.00     0.00  __gnu_cxx::__normal_iterator<char const*, std::__cxx11::basic_string<char,
  0.00      4.50     0.00   358986     0.00     0.00  __gnu_cxx::__enable_if<std::__is_integer<int>::__value, double>::__type std
```

Imagem 6.1 – Flat Profile do programa

Uma observação importante é que também foi feito um teste para um corpus de 18637 documento, presente no diretório “colecao_full/”, porém em um certo trecho do programa a execução demorava muito, com o tempo de execução chegando a horas e a execução não acabava. O trecho em que isso ocorreu foi o do cálculo das similaridades entre documentos e a consulta, de complexidade temporal O(n+i*(m+n*m)), como demonstrado anteriormente. Como para o corpus_full tanto o número de palavras do vocabulário (m, aproximadamente 22000), quanto o número de documentos no corpus (i, 18637) aumentam muito com

relação ao corpus_small, a complexidade temporal aumenta substancialmente e o programa começa a ter um tempo de execução enorme.

7. Conclusão

Esse trabalho foi uma ótima experiência. Com ele foi possível me aprofundar mais em vários assuntos da linguagem (C++) e adquirir mais habilidade com a linguagem. O uso de conceitos como alocação dinâmica de vetores, strings, uso de arquivos (principalmente a biblioteca systemfile, que nunca tinha sido usada anteriormente) e tratamento de documentos e textos, o que também foi uma novidade, com certeza aumentaram a minha perícia com a linguagem. Além disso, conceitos como Processamento de Linguagem Natural, Motor de Busca e a estrutura de mapeamento Hash foram introduzidos e utilizados na prática.

Com relação às dificuldades, eu poderia citar a preocupação com o gasto de tempo e de memória do programa, ou seja, sua eficiência, o que foi problemático em toda a implementação do código. Pela primeira vez esses conceitos que tanto estudamos no curso se mostraram reais e ficou claro o porquê da preocupação com a complexidade de tempo e espaço de um programa e como isso afeta até o computador onde o programa está sendo executado. Com certeza foi uma experiência muito engrandecedora, e as dificuldades foram importantes para o real entendimento do problema.

Bibliografia

ALGORITHMS FOR COMPETITIVE PROGRAMMING. **String Hashing**. Disponível em: https://cp-algorithms.com/string/string-hashing.html. Acesso em: 18 fev. 2022.

CPLUSPLUS. **Realloc**. Disponível em: https://www.cplusplus.com/reference/cstdlib/realloc/. Acesso em: 18 fev. 2022.

CPLUSPLUS. **Std::string::max_size**. Disponível em: https://www.cplusplus.com/reference/string/string/max_size/. Acesso em: 18 fev. 2022.

CPPREFERENCE. **Std::filesystem::path::filename**. Disponível em: https://en.cppreference.com/w/cpp/filesystem/path/filename. Acesso em: 18 fev. 2022.

DELFTSTACK. **Como obter a Lista de Arquivos em Diretório em C++**. Disponível em: https://www.delftstack.com/pt/howto/cpp/how-to-get-list-of-files-in-a-directory-cpp/. Acesso em: 18 fev. 2022.

DELFTSTACK. **Dividir String em C++**. Disponível em: https://www.delftstack.com/pt/howto/cpp/split-string-in-cpp/. Acesso em: 18 fev. 2022.

DREAM.IN.CODE. **Realloc() With String Arrays - C++**. Disponível em: https://www.dreamincode.net/forums/topic/224137-realloc-with-string-arrays-c/. Acesso em: 18 fev. 2022.

GEEKSFORGEEKS. **Pair in C++ Standard Template Library (STL)**. Disponível em: https://www.geeksforgeeks.org/pair-in-cpp-stl/. Acesso em: 18 fev. 2022.

IME USP. **Letras com diacríticos**. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/accents.html. Acesso em: 18 fev. 2022.

IME USP. **Unicode e UTF-8**. Disponível em: https://www.ime.usp.br/~pf/algoritmos/apend/unicode.html. Acesso em: 18 fev. 2022.

MICROSOFT. **Aliases e typedefs**. Disponível em: https://docs.microsoft.com/pt-br/cpp/cpp/aliases-and-typedefs-cpp?view=msvc-170. Acesso em: 18 fev. 2022.

PASSEI DIRETO. **Tabela ascii**. Disponível em: https://www.passeidireto.com/arquivo/72312429/tabela-ascii. Acesso em: 18 fev. 2022.

STACK OVERFLOW. **C++ sizeof(string)**. Disponível em: https://stackoverflow.com/questions/3629301/c-sizeof-string. Acesso em: 18 fev. 2022.

STACK OVERFLOW. **Rewind an ifstream object after hitting the end of file**. Disponível em: https://stackoverflow.com/questions/28331017/rewind-an-ifstream-object-after-hitting-the-end-of-file. Acesso em: 18 fev. 2022.

THISPOINTER. **Converting a String to Upper & Lower Case in C++ using STL & Boost Library**. Disponível em: https://thispointer.com/converting-a-string-to-upper-lower-case-in-c-using-stl-boost-library/. Acesso em: 18 fev. 2022.

VIVA O LINUX. **COMO RESOLVER WARNING: MULTI-CHARACTER? [RESOLVIDO]**. Disponível em: https://www.vivaolinux.com.br/topico/C-C++/Como-resolver-4. Acesso em: 18 fev. 2022.

WAGNER GASPAR. **Como aumentar o tamanho de um vetor com a função realloc?**. Disponível em: https://wagnergaspar.com/como-aumentar-o-tamanho-de-um-vetor-com-a-funcao-realloc/. Acesso em: 18 fev. 2022.

WIKIPÉDIA. **Função Hash**. Disponível em: https://pt.wikipedia.org/wiki/Função_hash. Acesso em: 18 fev. 2022.

WIKIPÉDIA. **Motor de Busca**. Disponível em: https://pt.wikipedia.org/wiki/Motor_de_busca. Acesso em: 18 fev. 2022.

Apêndice A - Instruções de Compilação e Execução

Para executar o programa recomenda-se usar o Makefile disponibilizado na matriz do diretório do programa pois ele já compila, linka e executa um programa para as 10 consultas presentes no diretório consultas. Caso o número de consultas seja alterado é necessário editar esse Makefile. Editando as linhas 26 a 35 do Makefile podem ser alterados os nomes dos documentos que contém as consultas (termo logo após a flag "-i"), o nome do documento de saída que vai conter o ranking (após a flag "-o"), o diretório que vai conter o corpus com os documentos da base de dados (após a flag "-c"), e o nome do documento que contém as stopwords a serem ignoradas na pesquisa (após a flag "-s"). Podem ser adicionadas novas linhas como essa ou removidas algumas existentes para alterar quantas consulta serão feitas.

Para executar deve-se seguir os seguintes passo:

- Utilizando um terminal acesse o diretório "Erick_Sunclair_2020026877/TP".
- Execute o arquivo Makefile utilizando o seguinte comando: "make all".
- Serão criados no diretório especificado os arquivos com os rankings para cada consulta com o nome designado.
- Para apagar os objetos criados na compilação do programa e os documentos tratados (que ficam no diretório "corpus_tratado/") basta executar no terminal outro comando do arquivo Makefile: "make clean". Isso deve sempre ser feito antes de executar o make file novamente pois devemos apagar os documentos tratados da execução anterior para que não fiquem resíduos dos documentos da última execução.
- E podemos executar novamente o programa.