Documentação Trabalho Prático 2

Lucas Roberto Santos Avelar

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

lucasrsavelar@ufmg.br

1. Introdução

Esta documentação lida com o problema da leitura de um texto e ordenação de suas palavras, em conjunto com a quantidade de vezes que essa palavra aparece no texto. O objetivo é implementar um programa que receba um arquivo contendo uma determinada ordem lexicográfica a ser seguida e um texto, além de argumentos passados pela linha de comando (nome dos arquivos de entrada e saída e inteiros a serem utilizados durante a escrita do programa), analise todas essas entradas e variáveis e retorne no arquivo de saída a impressão de todas as palavras presentes no texto, ordenadas pela ordem lexicográfica do arquivo de entrada, e sua frequência. Para resolver o problema, foi escrito um código que utilizava de algoritmos vistos em sala, tais como QuickSort e InsertSort, para realizar a ordenação das palavras, bem como de funções auxiliares escritas para determinar qual ordem esses algoritmos devem seguir e para tratar os casos de caracteres proibidos.

2. Implementação

O código organiza-se principalmente em 2 blocos. Dentro dos arquivos, encontram-se alguns comentários sobre as operações de cada função. A seguir, há uma análise mais detalhada.

O primeiro bloco consiste nos arquivos “funcoes.hpp” e “funcoes.cpp” que reúnem, respectivamente, a declaração e a implementação das diversas funções fundamentais para a execução do programa. A função returnID recebe um char e um array de char e itera sobre esse array procurando o índice em que o char se encontra, retornando esse índice de forma a atribuir um ID para cada letra, o que ajuda a determinar qual palavra deve vir antes de qual. Caso o char não seja uma letra e, por

consequência, não esteja no array, a função retorna o valor ASCII do caratere. A função toLower recebe uma string e itera sobre ela transformando cada char em minúsculo, de acordo com as especificações do trabalho. As funções ehProibido e removeChar recebem, respectivamente, uma string e um char e retornam true caso esse parâmetro seja um dos símbolos proibidos nas especificações. A função ehMaior representa uma das mais importantes do programa, pois recebe duas string e um array de char contendo a nova ordem lexicográfica e retorna verdadeiro caso a primeira string passada como parâmetro (left) seja maior que a segunda (right), de acordo com a nova ordem. Para isso, ela cria dois inteiros IDleft e IDright que recebem o ID do primeiro char de cada string, e uma variável que recebe o tamanho da menor string. Caso ambos os IDs sejam iguais, a função entra em loop e atribui o ID do próximo char, até que os IDs sejam diferentes ou que se alcance o tamanho da menor string. No segundo caso, se ambas as strings forem iguais até o tamanho da menor e a menor seja a left, então retorna-se que left é maior que right. Fora desse loop, se IDleft é maior que IDright, retorna-se verdadeiro. A função Insercao representa uma implementação do algoritmo InsertionSort, adaptado para ser utilizado dentro do QuickSort. É válido ressaltar o uso da função ehMaior negada, visto que ao invés de ordenar inteiros do menor para o maior, queremos que o maior ID (a primeira letra da nova ordem lexicográfica) venha primeiro. A função Particao representa o algoritmo de partição do QuickSort, também adaptado para atender aos requisitos da especificação. Primeiro, ele declara um inteiro que recebe o tamanho da partição. Caso esse tamanho seja menor ou igual ao parâmetro passado na inicialização do programa, a partição é ordenada usando o Insercao. Do contrário, inicia-se uma nova partição escolhendo um pivô. Se a mediana de M for maior que o tamanho mínimo da partição, o pivô é escolhido arbitrariamente; se não ele é a mediana dos M primeiros elementos ordenados da partição. Finalmente, ocorre a implementação do QuickSort, mais uma vez atentando para a utilização do ehMaior negado. Finalmente, a função QuickSort é a utilizada no programa principal e que chama a primeira partição.

Já o segundo bloco corresponde ao arquivo “main.cpp”, onde está localizada a função main do programa. Primeiro, ela declara e inicializa as variáveis necessárias para o funcionamento do programa. Então, ela entra em laço para verificar da linha de comando quais são os parâmetros passados para arquivos de entrada e saída e para valores

da mediana de M e do tamanho da partição. Caso os inteiros não sejam passados como parâmetro, ambos são considerados como 1. Os arquivos de entrada e saída são abertos e da entrada lê-se o início do bloco da nova ordem. Um laço coloca em um array de char cada letra, transformando-a em minúscula primeiro. Lê-se o início do bloco de texto e é criado um array de strings, que é inicializado com espaços em branco e depois recebe as palavras da entrada, a partir de um laço que continua enquanto houver elementos para ler do arquivo. Caso haja um caractere proibido, ele é substituído por um espaço em branco; do contrário ele entra em um novo laço que remove o último caractere da string enquanto ele for proibido. Finalmente, a palavra é transformada em minúscula e o índice é incrementado. Um laço conta o número de palavras, o array de palavras é ordenado usando o QuickSort otimizado e outro laço inicia-se para imprimir as palavras no arquivo de saída. Caso a próxima palavra seja igual à atual, o contador de repetições é incrementado. Se não, a palavra é impressa junto do seu número de repetições. É impresso o marcador final, os arquivos são fechados e encerra-se o programa.

O programa foi testado em um computador com 4GB de memória RAM e processador intel Core i3, sistema operacional Windows 10 (mas a partir de um subsistema WSL Ubuntu), utilizando da linguagem C++ e com o compilador g++.

3. **Análise de Complexidade**

Complexidade de Tempo: dentro do arquivo “funcoes.cpp”, temos que as funções ehProibido e ehMaior possuem complexidade O(1), visto que possuem apenas um condicional e o retorno. A função returnID também apresenta complexidade constante de O(26), pois sempre itera sobre um array de 26 posições. A função toLower apresenta complexidade O(n), pois depende do tamanho da string passada como parâmetro. A função ehMaior possui complexidade variável; a chamada de returnID e da função min são de complexidade constante, mas o laço while pode ser tanto O(log n) quanto O(n); no primeiro caso quando não é necessário percorrer todo o tamanho da string e no segundo caso quando toda a string é percorrida. A função Insercao, por ser uma implementação do Insertion Sort, possui complexidade O(n) no melhor caso e $O(n^2)$ no pior caso, como discutido em sala. Já as funções Particao e QuickSort, por serem implementações do algoritmo

QuickSort, possuem complexidade O(n log n) no melhor caso e O(n^2) no pior caso, também discutido em sala.

Dentro do arquivo “main.cpp”, as maiores complexidades de tempo residem no loop que lê as palavras da entrada, pois depende do número de string, tendo complexidade O(n), no loop que imprime as palavras na saída, que também depende do número de palavras com complexidade O(n), e na chamada do QuickSort, cuja complexidade já foi discutida.

Complexidade de Espaço: as funções realizam as operações levando em conta um arrays de string alocados de forma estática e que possuem tamanhos variados, mas sempre constantes (o array de char com a ordem possui sempre 26 posições e o array de palavras possui sempre MAXTAM posições). Sendo assim, a complexidade de espaço é constante e dada por O(MAXTAM), considerando que sempre teremos MAXTAM > 26.

4. **Estratégias de Robustez**

As estratégias de robustez adotadas foram as seguintes:

- Considerar os parâmetros da mediana e do tamanho da partição como 1, caso eles não sejam indicados na linha de comando ou sejam inválidos (negativos)
- Considerar que a saída será dada num arquivo denominado “saida.txt”, caso o parâmetro de saída não seja informado
- Tomar um pivô arbitrário caso mediana de M > tamanho da partição

5. **Testes**

Os testes foram realizados com auxílio dos arquivos de entrada disponibilizados no Moodle. Os 10 arquivos de entrada, bem como variações (incremento de palavras, adição de símbolos proibidos, entre outros) desses arquivos, foram fornecidos ao programa em conjunto com diferentes valores para a mediana de M e do tamanho da partição.

6. **Análise Experimental**

A análise de Desempenho Computacional foi efetuada após submeter o programa a entradas com diferentes números de palavras e com diferentes parâmetros de mediana e partição passados na linha de comando. Para números pequenos de palavras, o arquivo de saída do gprof não era capaz de produzir dados analisáveis do tempo empregado por cada função. Aplicando, então, uma entrada com 6500 palavras, foi possível verificar melhor quais funções consumiam mais tempo. Alterando os valores de partição e de mediana, algumas conclusões foram possíveis:

- quanto maior o tamanho da mediana, maior o equilíbrio dentro do tempo empregado pelas funções
- quanto menor o tamanho da partição, maior o equilíbrio também
- em todos os casos, a função returnID é a que mais consome tempo, muito por causa do extremo número de chamadas dessa função (mais de 685 mil no caso abaixo, mais do que 6x o número de chamadas da segunda função)
- mesmo com número absurdo de chamadas, o tempo foi extremamente curto, principalmente por nenhuma função apresentar complexidade elevada (a pior delas é $O(n^2)$)
- outras funções que demandaram a maior parte do tempo foram ehMaior e Particao

Abaixo registro de uma execução com 6500 palavras, partição de tamanho 1 e mediana de 50 elementos:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 60.08      0.03     0.03   685966     0.00     0.00  returnID(char, char*)
 20.03      0.04     0.01   104662     0.00     0.00  ehMaior(std::__cxx11::basic_string<char, std::char_traits<ch
 20.03      0.05     0.01        1    10.01    50.07  Particao(std::__cxx11::basic_string<char, std::char_traits<c
  0.00      0.05     0.00   104662     0.00     0.00  unsigned long const& std::min<unsigned long>(unsigned long c
  0.00      0.05     0.00    77000     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_stri
  0.00      0.05     0.00    64000     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_stri
  0.00      0.05     0.00    64000     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_stri
  0.00      0.05     0.00    58519     0.00     0.00  bool std::operator==<char, std::char_traits<char>, std::allo
  0.00      0.05     0.00    38500     0.00     0.00  bool __gnu_cxx::operator!=<char*, std::__cxx11::basic_string
  0.00      0.05     0.00    32000     0.00     0.00  toLower(std::__cxx11::basic_string<char, std::char_traits<ch
  0.00      0.05     0.00    13002     0.00     0.00  bool std::operator!=<char, std::char_traits<char>, std::allo
  0.00      0.05     0.00     8100     0.00     0.00  removeChar(char)
  0.00      0.05     0.00     6500     0.00     0.00  ehProibido(std::__cxx11::basic_string<char, std::char_traits
  0.00      0.05     0.00     6500     0.00     0.00  toLower(std::__cxx11::basic_string<char, std::char_traits<ch
  0.00      0.05     0.00     6500     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_stri
  0.00      0.05     0.00     6500     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>:
  0.00      0.05     0.00     6447     0.00     0.00  std::char_traits<char>::compare(char const*, char const*, un
  0.00      0.05     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z8returnIDcPc
  0.00      0.05     0.00        1     0.00     0.00  _GLOBAL__sub_I_main
  0.00      0.05     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.05     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.05     0.00        1     0.00    50.07  QuickSort(std::__cxx11::basic_string<char, std::char_traits<
```

7. **Conclusões**

O trabalho consistiu na implementação de um programa que recebe um arquivo de entrada contendo uma ordem lexicográfica a ser seguida e um texto, ordena essas palavras de acordo com a ordem por meio da implementação de um algoritmo QuickSort otimizado e imprime as palavras ordenadas, em conjunto com o número de vezes em que aparecem, em um arquivo de saída. Acredito que o trabalho em si tenha sido um pouco mais fácil de ser realizado do que o anterior. Em vários momentos foi possível reaproveitar recursos de diversas fontes para a escrita do código, o que não ocorreu no TP1. Além disso, a quantidade de exceções e casos particulares foi significativamente menor, o que torna a confecção do trabalho mais simples e menos propensa a erros. Acredito que o maior desafio tenha sido a implementação do algoritmo QuickSort otimizado, visto que foram necessárias algumas alterações para que fosse possível implementar as melhorias exigidas e em alguns casos era bastante difícil de entender o que deveria ser feito ou o que estava dando errado no processo de trocar de um método de ordenação para outro. Outro desafio foi ter de escolher entre uma alocação dinâmica ou estática para o array de palavras. Acredito que uma alocação dinâmica poderia reduzir bastante o espaço utilizado pelo

programa, já que não seria necessário alocar um vetor estático muito grande que por vezes não será completamente utilizado, o que acaba desperdiçando memória. Além disso, implementando uma estrutura própria para as palavras seria mais fácil contar o número de repetições de cada uma, já que poderia apenas criar uma variável que conta o número de vezes que a palavra aparece. Porém, o fator crucial para minha escolha de um array estático foi a possibilidade de trabalhar com índices. Os algoritmos de ordenação vistos na disciplina fazem forte uso dos índices dos arrays para procurar e trocar elementos, o que não é possível de ser feito em uma Lista Encadeada, por exemplo. Para mim, a complexidade de implementar algoritmos que não trabalhassem com índices não vale a economia de espaço, então preferi alocar um array estático de tamanho elevado, mesmo que parte dele seja desperdiçado.

8. **Bibliografia**

- Material disponibilizado no minhaUFMG
- Código proveniente das PA's anteriores (beecrowd)
- https://m.cplusplus.com/articles/DEN36Up4/
- https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/
- https://stackoverflow.com/questions/43956124/c-while-loop-to-read-from-input-file
- https://stackoverflow.com/questions/313970/how-to-convert-an-instance-of-stdstring-to-lower-case
- https://stackoverflow.com/questions/2310939/remove-last-character-from-c-string
- https://www.guru99.com/cpp-file-read-write-open.html

9. **Instruções para compilar e executar**

- Antes de tudo, **favor acessar o diretório "bin", presente dentro do diretório TP, e colocar nele o(s) arquivo(s) de entrada**
- Retorne ao diretório raiz TP
- Escreva e execute no terminal o comando "make"
- Acesse novamente o diretório "bin"
- Escreva e execute no terminal o comando "./tp2.exe", em conjunto com os argumentos da linha de comando. Isso significa:

  [-i/-I] nome do arquivo de entrada.txt **(Favor usar arquivos .txt!)**

  [-o/-O] nome do arquivo de saída

  [-s/-S] tamanho da partição em que será usado outro método de ordenação

  [-m/-M] mediana de quantos elementos serão utilizados na escolha do pivô

  Exemplo de entrada: ./tp2.exe -i input.txt -o saida.txt -s 3 –m 5

Observação: caso os parâmetros de mediana e partição não sejam informados, ambos serão considerados como 1. Caso o parâmetro de arquivo de saída não seja informado, ele será considerado como "saida.txt". É fundamental informar o arquivo de entrada!

A ordem em que os parâmetros serão passados não é relevante, desde que o indicador esteja seguido do parâmetro correto. Ou seja, para o exemplo acima:

./tp2.exe –m 5 -o saida.txt -s 3 -i input.txt também funciona

./tp2.exe -I input.txt –M 5 -O saida.txt -S 3 também funciona

./tp2.exe -s 3 -O saida.txt –M 5 -i input.txt também funciona

./tp2.exe -i input.txt -o 3 -s saida.txt –m 5 **NÃO!**

Também não faz diferença usar letras maiúsculas ou minúsculas nos indicadores, nem misturar maiúsculas com minúsculas.

- O arquivo de saída será gerado, normalmente, dentro do diretório "bin"

- Para novas entradas, acesse o diretório “bin” e execute novamente a linha de comandos