Documentação do Trabalho Prático 2

Luísa Ribeiro Bezerra

Junho, 2017

1. Introdução

O objetivo principal deste trabalho é trabalhar nossa perspectiva e aprendizado sobre a Memória Externa (também conhecida como Memória Secundária), além da manipulação da mesma com limite de uso de memória. Foi-nos dado um problema relacionado com a ordenação externa – no caso, implementei o Quicksort Externo – para a criação de um índice ordenado de palavras.

O contexto do Trabalho Prático é baseado na organização de conversas que o Hetelberto possui. Foi sugerido a criação do índice invertido com as palavras dos assuntos que estavam nas conversas dadas como arquivos para serem acessados. O índice invertido é uma estrutura de índice que armazena registros que mapeiam um conteúdo em documentos de uma coleção. Tais índices servem principalmente como forma de se realizar buscas rápidas em uma base de textos muito grande. O conteúdo geralmente é textual, ou seja, mapeia-se texto em documentos. Em uma coleção de documentos extremamente grande, torna-se inviável a busca textual usando algoritmos tradicionais de processamento de caracteres e faz-se necessário uma estrutura para facilitar tal busca. Uma implementação simples de um índice invertido - a qual foi utilizada neste trabalho prático - consiste construir registros no formato $< w,d,f,p >$, onde "w" é uma palavra, "d" é o documento onde encontramos aquela palavra, "f" é a frequencia da palavra naquele documento e "p" é a posição em bytes daquela palavra no documento.

Estes registros são gerados para todas as palavras no texto e ordenados em um grande arquivo. Este arquivo servirá como entrada para uma máquina de buscas. Durante a realização deste trabalho, é implementado, para a resolução do caso, a ideia de Quicksort Externo com uma memória interna limitada, o que nos obrigou a ter uma manipulação detalhada dos bytes possíveis de uso. Várias outras estruturas e funções foram aplicadas, como structs, ponteiros, arquivos binários, conversões e manipulação de arquivos.

2. Desenvolvimento

Para a solução do problema, comecei o fluxo do código lendo a entrada dada no arquivo de entrada, o qual contém uma linha contendo 2 números inteiros, que são o número de conversas e o tamanho limite de bytes do celular de Hetelberto, e logo após os caminhos para diretórios de entrada e saída, respectivamente. O diretório conterá conversas que serão arquivos numerados de 1 até D, onde cada arquivo será composto de texto puro com caracteres ASCII minúsculos (a-z) e espaços em branco (espaço e quebra de linha). Como dado no problema, as palavras não serão maiores do que 20 caracteres. No caso, guardo estas informações em dois inteiros, chamados numero_conversas e limite_palavras, e também em duas strings, chamadas entrada e saída. Para meu uso de manipulação de dados, crio um arquivo txt e um arquivo binário. Uso os caminhos de diretório de entrada e saída para a leitura da entrada e a criação da saída. Para tal, faço um loop entre as conversas para a leitura delas – a cada conversa aberta, leio a conversa e preencho as palavras com seus respectivos dados extras, que são a palavra, o número da conversa e a posição em bytes da mesma (a frequência, por ora, ainda não é atribuído valor a ela). Após todas as iterações, leremos todos os arquivos e teremos todas as palavras no arquivo txt, porém todas estão embaralhadas, uma vez que nenhuma ordenação ainda foi feita, somente foi lido as palavras e seus atributos. Para a ordenação, utilizei o arquivo binário, então antes de começar o método da ordenação, leio linha por linha do arquivo txt e os quebro em tokens [os atributos], os quais são repassados para o arquivo binário, sendo escritos por meio da função fwrite. Inclusive, a utilização do arquivo binário é justamente para facilitar a implementação do algoritmo, uma vez que por meio das funções fseek, fwrite e fread eu tenho um maior manuseamento dos arquivos. No próximo passo, há a chamada para a função qsexterno_inicializacao, que faz todo o trabalho de ordenação externa e interna do programa.

Para o qsexterno_inicializacao, utilizo uma struct (declarada em outro módulo, na biblioteca "funcoes_auxiliares.h") chamada PALAVRA, a qual encapsula a palavra binária (palavra lida no arquivo) e seus respectivos dados, como em qual conversa está, qual sua frequência e sua posição no arquivo em bytes. Crio uma string chamada pivô (limite de bytes total – o tamanho de um registro, que é 32 bytes, para o comparador ser ali alocado) e comparador para enviar para minha função de Quicksort Externo. Esta é a função base de todo programa para ordenar as palavras no arquivo de saída, e ela conta com seis marcadores, que são o LeituraEsq, LeituraDir, EscritaEsq, EscritaDir, direita e esquerda. Inicialmente, começo a leitura a fim de preencher o pivô por inteiro, sendo assim leio do arquivo binário as palavras com seus respectivos atributos e coloco-os em meu pivô, lendo de esquerda e direita alternadamente – entretanto, quando temos o encontro do leitor e do escritor no mesmo local, o código obriga a leitura daquele lado, uma vez que não podemos sobrescrever uma palavra ainda não lida. Após preencher (e a cada vez que mexo no pivô), ordeno ele com a ordenação interna qsort, que usa uma função de comparação que criei que considera inicialmente a palavra (caso elas sejam iguais ou não, usando do strcmp), após isso a comparação do número da conversa e após isso a comparação da posição da palavra. O qsort então faz a ordenação interna no pivô. Em cada leitura, a palavra lida do arquivo binário é comparado com o pivô - consideramos se ela está antes do pivô - e então escrevemos ela na esquerda -, depois do pivô - escrevemos ela na direita -, e caso esteja dentro do pivô - confiro se é maior ou menor do que

ou o último, retiro o maior ou o menor elemento do pivô de acordo com o lado que está maior (esquerda ou direita) para manter uma harmonia de tamanhos e escrevo no arquivo, e reordeno o pivô. Então, o conteúdo arquivo é particionado e toda a lógica é refeita até chegar no ponto que a partição tem o tamanho do pivô, então ele é inserido exatamente no lugar certo, ordenado, e a lógica de ordenação está completa.

O arquivo binário agora está ordenado, somente faltando a atribuição da frequência de cada palavra. Para tal, chamo, após a chamada da função qsexterno_inicializacao, a função calcula_frequencia, a qual recebe o arquivo binário e o número de palavras existentes e faz um loop por entre elas procurando a frequência de palavras em cada conversa. A ideia de já ter um arquivo ordenado para a contagem de frequência é pelo fato de ela ficar menos custosa, uma vez que as palavras estão sequencialmente posicionadas, linha após linha, somente diferindo a conversa que estão e a palavra a ser analisada. Nesta função, leio uma palavra, e caso não tenha atingido o final do número de palavras deste loop, leio a próxima palavra. Verifico se a palavra seguinte é a mesma da palavra atual lida, e caso não sejam, o loop é quebrado e a leitura de palavras reinicializa com o contador de frequência reinicializado. Caso sejam iguais, comparo, então, o número da conversa que estão. Caso não sejam iguais, quebro o loop e reinicializo também o contador de frequência. Caso sejam iguais, o contador de frequência é acrescentado de 1, e o loop continua a leitura da próxima palavra, até encontrar uma que quebre o loop batendo com as condições de quebra explicadas. Toda vez que há a quebra de loop, preencho e atualizo os valores das frequências até onde foi verificado a igualdade entre elas no arquivo binário.

Por fim, há a conversão do arquivo binário em arquivo txt final. Utilizo aqui um buffer que transita as palavras do arquivo binário para a escrita delas no arquivo txt. Este buffer irá ler do arquivo binário a palavra e escrever no arquivo txt. Faço isso com todas as palavras, que já se encontram ordenadas e com seus dados preenchidos corretamente. Após ser escrito, o arquivo binário é deletado por meio de um remove("bin").

3. Análise de Complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço dos principais algoritmos e estruturas de dados utilizados. Portanto, foram analisadas as funções abaixo contidas na tabela:

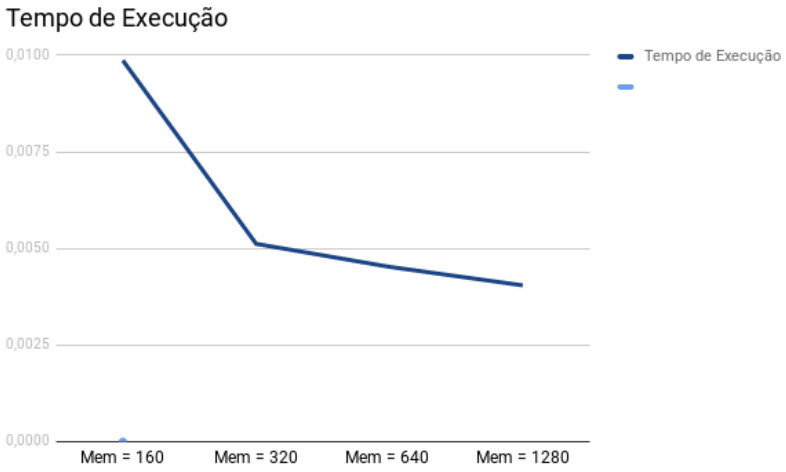
Quicksort Externo	O Quicksort Externo depende da memória disponível fornecida na entrada do programa; no melhor caso, a complexidade é O(n/b). No pior caso, a complexidade é $O(n^2/m)$, no caso médio é O((n/m)*log(n/m)), Sendo n = número de palavras a serem ordenadas, e b = tamanho do arquivo de entrada e m= memória primária disponível.
qsort	A complexidade do qsort, que é implementado pela biblioteca padrão do C, é O(n*log(n)).
Cálculo da Frequência	Temos, neste cálculo, a existência de três (03) loops encadeados, o que torna, no pior caso, a complexidade deste cálculo de $O(n^3)$.

A **complexidade total do programa** é a maior complexidade dentre as citadas acima, que é a do Quicksort Externo, pois ele faz acesso à memória secundária e isso é um fator primordial para tornar lento o programa.

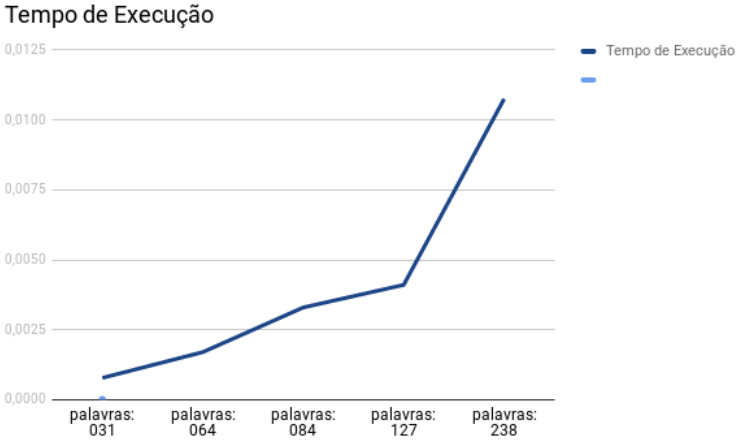
A **complexidade espacial** depende do número de palavras a serem ordenadas e da memória interna disponível, dado nas entradas fornecidas ao programa.

4. Avaliação Experimental

Os experimentos foram executados em um sistema com 8 GB de memória RAM e um processador Intel Core i7-3632QM CPU @ 2.20GHz. Para os testes, criei dois gráficos baseado no tempo de execução (segundos) versus o número limitante de bytes, e o outro é baseado no tempo de execução versus o número de palavras. Sempre que repeti o teste para a mesma entrada, o tempo de execução foi bem aproximado, o que mostra que o algoritmo é bem estável. Foram repetidas 10 vezes cada teste e feito a média deles.



Neste teste acima, o número de palavras estava fixo em 228 palavras.



O limite de memória está fixo em 160 bytes no teste acima.

1. Conclusão

Sempre que realizei os testes, foi possível analisar que o crescimento do tempo de execução foi inversamente proporcional ao aumento do limite de bytes (que é fornecido como entrada) e ao aumento do número de palavras. Isso conclui que a complexidade de espaço, dependente da entrada, é um fator primordial para o tempo de execução, sendo alto para pequenos limites, e baixo para um limite alto (inversamente proporcional).

O trabalho foi fundamental para a aplicabilidade de teorias que aprendi durante o curso - implementação de uma ordenação externa, manipulação de arquivos e uma lógica bem avançada de controle de uso de memória interna. A utilização do qsort também foi interessante para ampliar a ideia da base de bibliotecas disponíveis que podemos usar, bem como a criação de uma função própria para ser enviada para ele. O trabalho prático trouxe um tempo muito aquém do esperado, visto a dificuldade do processo, e inclusive a entrega tardia. Achei a proposta muito interessante e pude, com toda clareza, desenvolver minhas habilidades de lógica e programação.