Algoritmos e Estrutura de Dados III
Ordenação Externa

Ingrid Rosselis Sant'Ana da Cunha

Junho 2017

1 Introdução

A ideia do trabalho prático 2 da disciplina Algoritmos e Estrutura de Dados III é ordenar arquivos que contenham registros, na forma de índice invertido, tendo apenas um tamanho limitado de memória para essa operação. A situação contada na história que apresenta este trabalho é na verdade um problema real em muitas situações onde todos os registros de um arquivo não cabem na memória. Um índice invertido é uma estrutura de dados que mapeia termos às suas ocorrências em um documento ou conjunto destes. É uma estratégia de indexação que permite buscas rápidas e ordenação de registros em uma memória de tamanho limitado. Para este trabalho, o índice invertido armazena a palavra, o documento a qual pertence, a frequência com a qual a palavra aparece no documento e a sua posição neste.
Um algoritmo de ordenação externa pertence a uma classe de algoritmos de ordenação que utilizam memórias nas categorias primária e secundária e que conseguem lidar com grandes quantidades de dados. Em casos reais, a memória primária é a memória RAM e a memória secundária pode ser um disco rígido, um disquete ou um servidor em nuvem. Para fins educativos, a memória deste trabalho limita a quantidade de registros vindos de arquivos que podem ser ordenados por vez. Após m, sendo m o tamanho da memória, registros serem ordenados dentro do espaço desta, eles devem ser reescritos em seu arquivo de origem e os blocos ordenados devem ser reordenados entre si, para que a execução do algoritmo de ordenação termine. Esta estratégia é amplamente utilizada por serviços de gerenciamento de banco de dados e por serviço de buscas.

2 Implementação.

Para a realização do trabalho foram utilizados três algoritmos: quicksort externo - para ordenação externa -, quicksort e insertionsort, estes dois últimos para ordenação interna. Para lidar com o número de funções e algoritmos requeridos o projeto foi dividido em oito arquivos, sendo eles:

1. Dois arquivos para funções gerais como tratamento da entrada e de arquivos texto e binário;
2. Dois arquivos para funções auxiliares à ordenação externa e para os algoritmos de ordenação interna;
3. Dois arquivos unicamente para a ordenação externa;
4. Um arquivo contendo a função principal;
5. Um makefile.

Quanto à implementação, o código como um todo pode ser resumido nesses passos:

1. Lê a entrada;
2. Abre os arquivos auxiliar e final;
3. Abre e lê as conversas;
4. Monta os registros com base no formato índice invertido;
5. Escreve os registros no arquivo auxiliar;
6. Ordena externamente o arquivo auxiliar;
7. Conta a frequência de cada palavra;
8. Copia o conteúdo do arquivo auxiliar para o arquivo final;
9. Fecha os arquivos utilizados.

Para a realização do trabalho foi preferido fazer toda a programação sobre um arquivo binário, pois os registros podem ser facilmente lidos e escritos sem a necessidade de serem quebrados. Assim, toda a parte de ordenação e contagem de frequência é feita em arquivo binário, e o arquivo final é utilizado unicamente para a entrega da solução, como especificado.
A comparação de registros para a ordenação lexicográfica foi parte importante da implementação de todos os algoritmos utilizados neste projeto e merece um destaque. Ela é feita baseada em quase todos os campos do registro, mais especificamente, sob a palavra, seu documento e sua posição dentro do documento. Para este fim, a função *Compare* compara dois registros da seguinte forma:

```
1: procedure COMPARE(dois registros)
2:     compara palavras
3:     if mesma palavra then
4:         compara documentos
5:         if mesmo documento then
6:             compara posições
7:             if mesma posição then
8:                 retorna 0
9:     caso contrário, retorna o menor registro
```

Para a ordenação externa do arquivo auxiliar foi utilizado o quicksort externo pois além de possuir um desempenho satisfatório, apenas usa quatro ponteiros como memória auxiliar, dois para controle de leitura e dois para controle de escrita, diminuindo o risco de estouro de memória. Ele funciona praticamente da mesma maneira que o quicksort interno, porém o pivô é substituído por um vetor de registros ordenados. Ele também age sobre a ideia de divisão e conquista, repartindo o arquivo em subarquivos para serem ordenados recursivamente.
O subarquivo gerado pelo quicksort externo é particionado e localmente ordenado da seguinte forma: a memória é preenchida com registros vindos de ambos os lados do arquivo, isto porque quatro variáveis e uma $flag$ garantem que haja intercalação sempre que houver uma leitura no arquivo e que a leitura e a escrita sempre ocorram uma posição a mais em direção ao centro do arquivo, forçando os dois lados a serem igualmente percorridos; depois de preenchida a memória em sua totalidade ou com o número de registros que existem no subarquivo, ela é ordenada usado o quicksort interno, pela simplicidade de implementção e porque há uma grande possibilidade de que ela não esteja ordenada nessa primeira iteração (diminuindo o risco de que o pior caso $O(n^2)$ se concretize).
Agora o algoritmo pode seguir para a parte da ordenação, que ocorre da seguinte maneira: um próximo registro é lido do início ou do fim do arquivo, dependendo de onde se encontra a intercalação, e em comparações feitas com esse registro, há três possibilidades de continuar o algoritmo:

1. O registro lido é menor que o menor registro armazenado na memória:
   Neste caso o registro lido é escrito no início do arquivo, mais especificamente onde se encontra a variável de controle da escrita para essa parte, e a execução segue normalmente.

2. O registro lido é maior que o maior registro armazenado na memória:
   Neste caso o registro lido é escrito no fim do arquivo, mais especificamente onde se encontra a variável de controle da escrita para essa parte, e a execução também segue normalmente.

3. O registro lido não é maior que o maior registro ou menor que o menor registro da memória:
   Neste caso o menor ou o maior registro da memória é escrito no arquivo, de acordo com a diferença entre os deslocamentos das variáveis de escrita, e o registro lido entra na memória na posição do maior/menor que foi retirado, pois a memória não possui controle próprio dos elementos que

a preenchem. A seguir, a memória é reordenada para continuar com a execução. Para esta última ordenação, é utilizado o insertionsort pois a memória está quase completamente ordenada, com exceção do último registro inserido.

Quando as váriaveis de leitura chegam ao meio do arquivo, os registros ordenados que estão na memória são escritos na posição da variável de escrita do início, ou seja, o algoritmo já insere um bloco ordenado dentro do arquivo e após um certo número de execuções, acaba por ordenar todo o arquivo.
O algoritmo recursivo do quicksort externo se encerra quando ocorre o encontro entre os lados esquerdo e direito no algoritmo, isto é, quando a partição resultante não é válida para ser ordenada e repartida.
Abaixo segue um pseudo-código resumindo o funcionamento do quicksort externo quando o arquivo é partido e ordenado em memória:

Algorithm 1 Quicksort externo

```
    procedure PartDisk
2:      leitura_inferior = escrita_inferior = começo do arquivo
        leitura_superior = escrita_superior = final do arquivo
4:      pré-define novas partições
        while memoria não cheia e leitura_superior ≥ leitura_inferior do
6:          lê registro
            memoria ← registro_lido
8:      ordena memória
        while leitura_superior ≥ leitura_inferior do
10:         lê um registro
            if registro_lido < menor_registro_memoria then
12:             redefine partição da esquerda
                arquivo ← registro_lido
14:         if registro_lido > maior_registro_memoria then
                redefine partição da direita
16:             arquivo ← registro_lido
            else
            if deslocamento_esquerda < deslocamento_direita then
18:             arquivo ← menor_registro_memoria
                memoria ← registro_lido
            else
20:         arquivo ← maior_registro_memoria
            memoria ← registro_lido
22:         reordena memória
        arquivo ← memoria
24: procedure ExternalQuicksort
        chama PartDisk
26:     chama recursões para parte da esquerda e da direita
```

Os algoritmos de quicksort interno e insertionsort já são conhecidos e foram estudados em outras disciplinas, com a diferença de que as comparações são feitas utilizando a função *Comapre*. Por fim, a contagem de frequência dos registros em um mesmo documento foi feita da seguinte forma:

```
1: procedure CONTAGEM DE FREQUÊNCIA(arquivo ordenado)
2:     abre arquivo em auxiliar
3:     define contador, frequência e nova_palavra (flag)
4:     nova_palavra ← 0
5:     pega primeiro registro
6:     while !fim_do_arquivo do
7:         lê registro
8:         if mesma palavra e mesmo documento then
9:             incrementa frequência
10:            nova_palavra ← 1
           else
11:        while contador < frequencia do
12:            volta para primeira palavra igual com auxiliar
13:            lê registro
14:            atualiza frequência
15:            escreve registro
16:        contador ← 0, nova_palavra ← 0, frequencia ← 0
17:        pega próximo registro já lido
18:    if nova_palavra == 1 then
19:        refaz processo de atualização para último bloco de palavras iguais
```

3 Análise de Complexidade

3.1 Espacial

A complexidade espacial para este trabalho é $O(m)$, onde m é o tamanho da memória passado como parâmetro, pois a ideia do projeto é que a ordenação se dê no espaço limitado da memória, sem gastos adicionais.

3.2 Temporal

Para a complexidade de tempo, o maior gasto se dá com o algoritmo de ordenação externa pois acessos à arquivos são mais caros que acessos à memória primária. Assim como o quicksort interno, o quicksort externo se faz valer com o paradigma da divisão e conquista, o que torna sua complexidade logarítmica no caso médio e quadrática no pior caso.

Caso Médio:

$$O(n * \log_2 n/m) \tag{1}$$

Pior Caso:

$$O(n^2/m) \tag{2}$$

Sendo n o número de registros a serem ordenados e m o tamanho da memória. O pior caso do algoritmo ocorre quando um dos subarquivos retornados pelo procedimento tem o maior tamanho possível e o outro é vazio.
A complexidade do quicksort externo acontece devido ao fato de que a cada passada do algoritmo o problema é dividido em dois, vezes n/m, já que a cada execução o pivô do algoritmo é um vetor de m posições, que divide o número de registros.
Quanto à complexidade dos algoritmos de ordenação interna, o insertionsort possui complexidade $O(n^2)$, mas é usado no caso em que a memória está quase ordenada, o que diminui seu custo, e o quicksort possui complexidade $O(nlogn)$. No entando, o custo para ordenar a memória é tomado como $O(1)$ já que essa ordenação é irrelevante perante aos numerosos acessos à memória secundária.
Por fim, pode-se afimar que o programa realiza o quicksort externo de maneira independente e que a complexidade dele é assintoticamente superior ao resto do programa. Então, por isso, pode-se dizer que a complexidade deste trabalho é a mesma complexidade que o quicksort externo.

4 Avaliação Experimental

Para realizar a avaliação experimental deste trabalho, foram utilizados 100 casos de teste feitos por colegas de classe. Foi constatado durante o experimento que o tempo não é influenciado por quantas conversas existem, mas sim pelo tamanho total delas. Por esse motivo, a análise leva em conta dois parâmetros:

1. Variação do tamanho da memória principal;
2. Variação do tamanho das conversas.

Os casos de teste foram gerados por *script* e utilizam a notação (x, y), onde x é o tamanho das conversas e y é o tamanho da memória, ambos variando entre 0 e 9 e medidos em bytes. Falando em números, o tamanho das conversas varia entre 100000 e 190000 e o tamanho da memória varia entre 100 e 9100. As fórmulas para gerar o tamanho das conversas e da memória em cada caso são:
Tamanho das conversas:

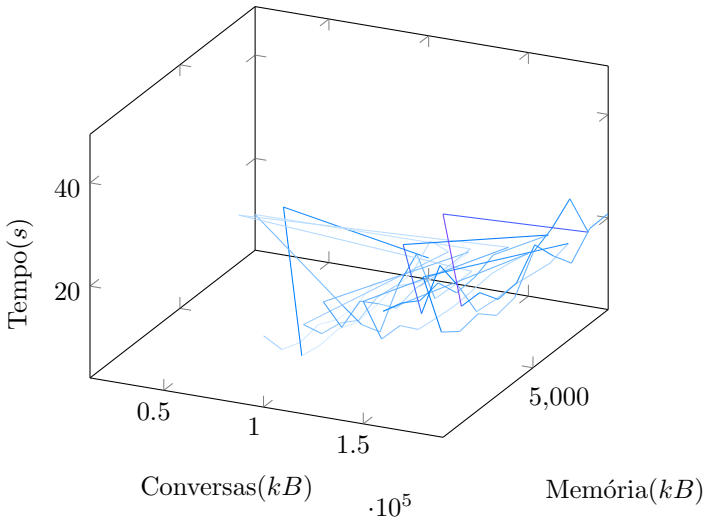
$$10^5 + 10^4 * x \tag{3}$$

Tamanho da memória:

$$10^2 + 10^3 * y \tag{4}$$

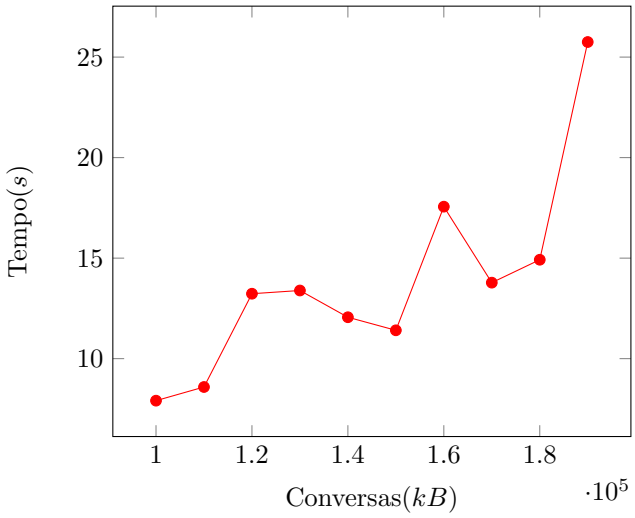
A quesito de curiosidade, o gráfico abaixo mostra a variação do tamanho da memória e das conversas juntas em relação ao tempo, apesar de não ser muito explicativo.

Gráfico conversa x memória x tempo



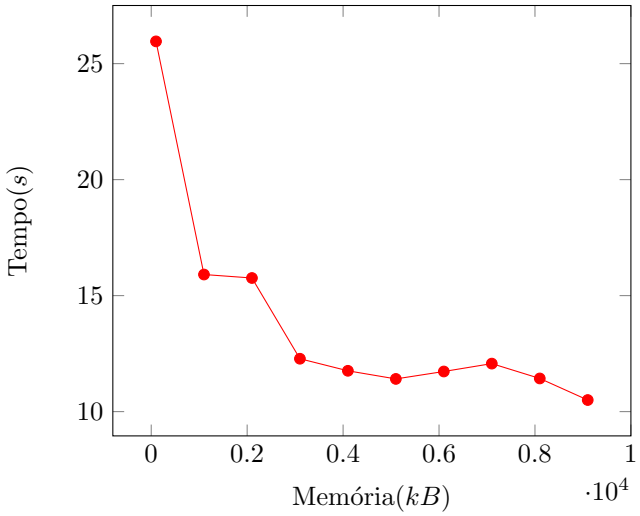
O gráfico abaixo foi feito separando 10 testes que foram usados no gráfico 3D e que possuem tamanho de memória fixo em 5100 e, como pode-se notar, o tempo de execução aumenta conforme o tamanho das conversas aumenta. Isso faz sentido pois a complexidade varia em função de n/m e mantendo-se fixo o m equanto n cresce, o resultado tende a ser maior:

Gráfico conversa x tempo



O próximo gráfico foi obtido também separando 10 testes que foram usados no gráfico 3D e que possuem o tamanho das conversas fixo em 150000. Neste caso, nota-se que o tempo de execução diminui conforme o tamanho da memória aumenta, pois a complexidade varia em função de n/m e o n é mantido fixo enquanto m cresce, então o resultado tende a diminuir:

Gráfico memória x tempo



5 Conclusão

O objetivo deste trabalho é implementar um algoritmo de ordenação externa com um tamanho de memória limitado. De modo geral, o algoritmo teve um funcionamento de acordo com o esperado, atendendo aos requisitos pré-especificados e obtendo resultados satisfatórios. Por fim, este trabalho reforçou o entendimento acerca do uso de arquivos e clarificou a ideia do gasto de memória por algoritmos, já que previamente a preocupação era, como um todo, apenas com o gasto de tempo dos algoritmos. Ficou explícito, então, que nem sempre a solução para um problema pode ser obtida com maior eficiência sacrificando memória primária em prol do desempenho.