Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

DCC005 - ALGORITMOS E ESTRUTURAS DE DADOS III - TURMA TE

Trabalho Prático 3 – Facilitando a correção

Guilherme Saulo Alves

< guisaulo@hotmail.com >

Professores - Jussara Marques, Olga Nikolaevna,
Marcos Augusto, Wagner Meira

1. Introdução

A ideia do trabalho consiste em realizar um índice invertido dos termos nas respostas dos alunos que fizeram uma dada prova. Os termos nas repostas de cada aluno forma um documentos *da* em que *a* é um identificador do documento referente ao aluno. Um índice invertido é uma estrutura de dados para indexar uma coleção de documentos *D*. Para cada palavra pertencente a *D*, tem-se uma lista de alunos que usaram esse termo na coleção.

Para produzir o índice invertido é preciso saber as palavras no conjunto *D* e ordená-los. Essas informações foram representadas em tuplas <*t* , *a*> em que *t* é um termo pertencente a uma resposta do aluno *a*. Com essa representação, criamos uma lista de tuplas *L* e ordenamos para produzir o índice invertido. No entanto, para o presente trabalho a lista *L* é grande demais para se ter na memória, o que impossibilita sua ordenação utilizando apenas a memoria interna. Logo, foi necessário utilizar um algoritmo de ordenação externa para produzir a lista ordenada *L* e produzir o índice invertido.

Levando-se em consideração esses fatos, foi implementado o algoritmo de ordenação externa ***Intercalação por Substituição***. Na primeira fase do algoritmo, a lista *L* a ser ordenado é quebrada em blocos que são ordenados na memória principal. Um bloco é então escrito em uma fita de saída e um novo bloco é ordenado. Esse processo continua até que todos os termos presente na lista estejam nos blocos ordenados. A segunda fase da ordenação consiste em intercalar esses sub-blocos ordenados para que gerem a lista *L* final ordenado. O algoritmo será explicado com detalhes ao longo da documentação.

2. Implementação

O ambiente de desenvolvimento do código foi via compilador GNU GCC Compiler via linha de comando no Sistema Operacional Ubuntu Linux 13.04.

Para gerar o programa executável, utiliza-se o comando *make* no shell do Linux. O programa gerado possui nome *tp3* e deve aceitar, dependendo da ordem, um dos seguintes argumentos:

```
./tp3 -o <arquivo de saída> -m <tamanho da memória> <arquivo(s) de entrada>...
./tp3 -m <tamanho da memória> <arquivo(s) de entrada>... -o <arquivo de saída>
```

O código está dividido em seis arquivos principais:
- **main.c:** Arquivo principal do programa, contém a chamada de execução da função principal;
- **io.c:** Realiza a comunicação do programa com o ambiente, lendo os arquivos de entrada, criando as fitas e escrevendo nos arquivos de saída;
- **ordenacao.h:** Ordena o conteúdo de um arquivo utilizando arquivos auxiliares, pelo método de Intercalação Balanceada;
- **heap.c:** Possui as funções de um Heap mínimo;
- **indice.c:** Produz o índice invertido a partir da lista ordenada;
- **biblioteca.h:** Possui os TADs e as declarações de todas funções usadas no programa.

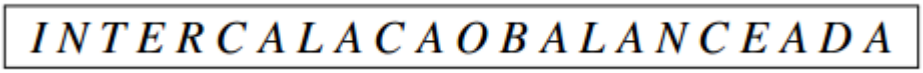
2.1. Funções, Estruturas e Algoritmos

As soluções dos problemas serão apresentadas a seguir, através de Pseudocódigos e esquemas ilustrativos. O variáveis e funções possuem nomes que facilitam a análise do código.

2.1.1. Considerações Iniciais

Para a explicação das funções e algoritmos a seguir considere os seguinte aspectos:

- Seja f o número de fitas para intercalação, a memória principal poderá armazenar no máximo f tuplas. Para o trabalho, foi utilizado um total de 8 fitas (4 para armazenar blocos de termos do arquivo e 4 com a intercalação dos blocos anteriores). Fica a escolha do usuário modificar a macro NUMFITAS 4 no código;
- As fitas no programa são arquivos de texto temporários com os seguintes nomes: Fita_0.txt, Fita_1.txt, Fita_2.txt, ... , Fita_2*f".txt, onde f é um inteiro que corresponder ao número de fitas;
- Para ilustrar o algoritmo de intercalação, utilizamos o esquema ilustrativo abaixo. Considere um exemplo de arquivo com 22 termos 6 fitas (3 para formação de blocos ordenados iniciais e 3 para intercalação das 3 fitas anteriores) abaixo:



2.1.2. Programa Principal

- **int main(int argc, char *argv[]);**

Contém a chamada da função principal. Para facilitar a compressão, podemos dividir o programa principal em quatro partes em um ilustração.

Na primeira parte, o programa verifica os argumentos de execução e inicializa algumas variáveis de controle. Na segunda parte, chama-se a função que cria um arquivo lista.txt com as tuplas, a partir dos arquivos de entrada. Na terceira parte, usamos o arquivo gerado na parte anterior para produzir um arquivo chamado lista_ordenada.txt quando chamamos a função de ordenação externa.

Por fim, na quarta parte, geramos o índice invertido num arquivo com o nome do arquivo de saída, a partir do arquivo ordenado na parte anterior.

2.1.3. Estrutura TipoHeap

A estrutura TipoHeap, simula um TAD heap onde armazena as linhas lidas do arquivo de entrada, marca os membros do próximo bloco e armazena o número das fitas (arquivos temporários).

2.1.4. Funções de entrada e saída

- **int CriaLista(short inicio, short fim, char *argv[]);**

Lê todos os arquivos de entradas e escreve as tuplas no arquivo lista retornando a quantidade de termos. Será imprenso um termo e aluno por linha no arquivo lista.txt. Avisa se o arquivo foi gerado corretamente.

- **short CriaFitas(TipoHeap *Heap, FILE *ArqEntrada, FILE **ArqTemporario, short MaiorLinha);**

Cria as fitas e faz a impressão nos arquivos temporários. Foram criadas 8 fitas, 4 para entrada e 4 para saída.

- **void RemoveArqTemporarios(FILE** ArqTemporario, short TotalFitas);**

Função para fechar e remover os arquivos temporários do diretório. A função remove está como comentário para visualizar as fitas no diretório. Caso queira remover os arquivos, retire os comentários.

2.1.5. Funções de ordenação

- **void OrdenacaoExterna(FILE *ArqEntrada);**

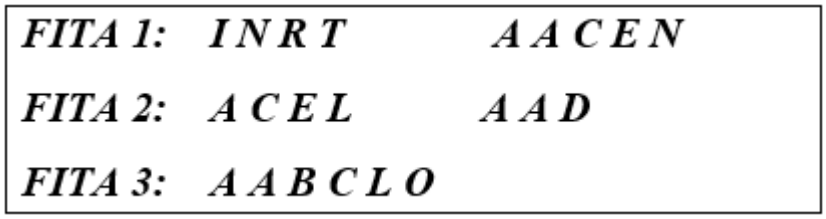
A função realiza a primeira fase do algoritmo de intercalação que corresponde a primeira passada no arquivo de entrada (lista.txt) para criar os blocos ordenados para a segunda fase de intercalação.

O primeiro procedimento é passar pelo arquivo de entrada armazenando f tuplas no heap. A menor tupla é retirada do heap e armazenado em uma das primeiras fitas. Sucessivamente, outra tupla do arquivo de entrada é inserido no heap.

A tupla que foi inserida é comparada com a tupla que foi retirada. Se a tupla que foi inserida for menor que a tupla que foi retirada, essa tupla será marcada. A marcação indica que a tupla é o maior elemento que está no heap. Posteriormente, o heap é refeito.

Quando todos os elementos do heap estiverem marcados, um bloco é feito e separado por um ***'\n'***. Os campos de marcação são zerados e um novo bloco é criado. O novo bloco é imprimido na segunda das f fitas. Ao chegar na última fita f, o próximo bloco será colocado na primeira fita novamente, e assim sucessivamente.

Esse procedimento é feito até ler todas as tuplas do arquivo de entrada. No final do procedimento, tem-se f fitas em que cada uma tem 1 ou mais blocos. A figura abaixo ilustra a formação dos blocos ordenados iniciais para o exemplo anterior:



- **void Intercalacao(TipoHeap *Heap, FILE **ArqTemporario, char *NomeSaida, short NumDeBlocos1, short MaiorLinha);**

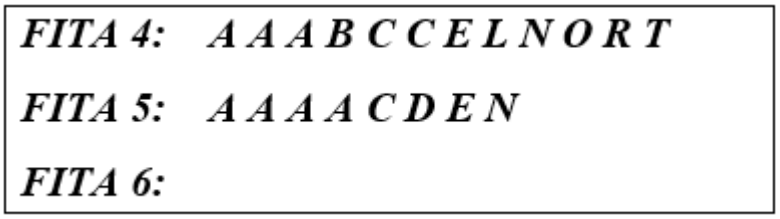
A função faz a segunda fase da intercalação entre as fitas pelo método Seleção por substituição.

Um novo heap será criado com as primeiras tuplas dos primeiros blocos de cada fita. A menor tuplas será retirada e colocada na primeira fita restante. Outra tupla será inserida e o heap é refeito. Essa nova tupla virá da

fita em que a última tupla retirada. Esse processo é feito até que chegue ao fim de todos os primeiros blocos das primeiras fitas.

Terminando o passo anterior, outra intercalação é feita com os segundos blocos das primeiras fitas e o resultado é enviado para a segunda fita restante. O processo continua até que todos os blocos sejam intercalados.

No caso em que chegamos ao fim da última fita restante e ainda restar blocos para serem intercalados, o resultado volta a ser enviado a partir da primeira fita restante, com a delimitação ***'\n'*** no fim dos blocos, assim sucessivamente. O procedimento acaba quando os blocos contenha todos os elementos ordenados e a quantidade de blocos criados na intercalação é igual a 1. A figura abaixo ilustra a intercalação-de-3-caminhos usando seleção por substituição:



2.1.6. Funções do heap

- **void ConstroiHeap(TipoHeap *A);**

Constrói um Heap com os elementos das linhas do arquivo. Essa função calcula os parâmetros para a função RefazHeap.

- **void RefazHeap(Indice Esq, Indice Dir, TipoHeap *A);**

Refaz a condição do heap caso alguma tupla saia ou substitua outra no vetor. O menor elemento do heap estará no início do heap, Heap[1].

- **void RetiraMinHeap(TipoHeap *A, FILE *ArquivoEntrada, short MaiorLinha);**

Função para retirar o menor elemento do heap, ou seja, o primeiro elemento do heap.

3. Analise de Complexidade

A solução proposta consiste na implementação das funções CriaLista, Ordenacao (OrdenacaoExterna + Intercalacao) e IndiceInvertido. Para a análise de complexidade de tempo e espaço da solução implementada considere os seguinte aspectos:

- Seja ***f*** o número de fitas de entrada intercaladas, ***n*** o número de termos do arquivo de entrada, ***t*** tamanho máximo dos termos lidos dos arquivos de entrada e ***m*** o tamanho do buffer do heap.

3.1. Complexidade CriaLista

- **Complexidade de Espaço:** O principal uso da memória consiste em alocar um vetor de strings para ler os termos dos arquivos de entrada. Logo a complexidade de espaço é **O(*t*).**
- **Complexidade de Tempo**: O principal consumo de tempo está no loop que ler todas linhas do arquivo de entrada até seu fim. Logo a complexidade de tempo é **O(*n*).**

3.2. Complexidade Ordenacao

- **Complexidade de Espaço:** O principal uso da memória consiste em intercalar os termos no buffer do heap. Logo a complexidade de espaço é **O(*f*).**
- **Complexidade de Tempo:** O principal consumo de tempo está nos 2 loops. O primeiro loop possui dependência de f, logo O(f). A partir do segundo loop, temos o algoritmo do método Intercalação de Seleção por Substituição. O tamanho do buffer do heap é igual à quantidade e fitas de entrada ***m=f*.** A complexidade de tempo fica O(logf(n/m))[1]. Logo a complexidade total de tempo total é **O*(f)* + O*(logf(n/m))*.**

3.3. Complexidade IndiceInvertido

- **Complexidade de Espaço:** O principal uso da memória consiste em alocar vetores de strings para ler os termos dos arquivos de entrada. Logo a complexidade de espaço é **O(*t*).**
- **Complexidade de Tempo:** O principal consumo de tempo está no loop que ler todas linhas do arquivo ordenado até seu fim. Logo a complexidade de tempo é **O(*n*).**

4. Experimentos

Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória.

4.1. Tabela

Para avaliar o funcionamento do programa, utilizou-se vários tamanhos de entradas com termos aleatórios. Abaixo segue uma tabela com as respectivas analises:

Tamanho Arquivo	Máx. Mem. Atingida	Quantidade de termos	Tempo Sistema em segundos	Número de Fitas
140 bytes	632	13	0,009	8
2 megabytes	712	221540	0,152	8
10 megabytes	712	2204269	0,644	8
70 megabytes	764	11022007	9,972	8
130 megabytes	632	22043746	11,519	8
1 gigabytes	708	180745719	32,568	8

4.2. Observações importantes sobre os testes

- O parâmetro “tamanho da memória” serve pra dizer que existe um limite para memória interna durante o processo. Embora seja passada como parâmetro, a quantidade de memória não foi usado pelo programa visto que a quantidade de fitas já diz quantas tuplas a memória interna irá armazenar.
- O programa tem usa no maximo 764 Kbytes de memória principal, logo a quantidade ideal de memória passada como parâmetro deve ser superior a 764 Kbytes.
- Foi usado o comando ***time /usr/bin/time –f “%M” ./programa*** para fazer os testes de memória e tempo do sistema.
- Devido ao tempo corrido, não foi possível elaborar os gráficos com as respectivas analises. Pelo enunciado do TP, supus que somente as palavras devem estar em ordem, porém quando fui entregar o trabalho, percebi que os números dos alunos também estavam ordenados. Somente alguns testes com o número de aluno maior que 2 algoritmos deu diferente. Porém o programa rodou perfeitamente.

5. Conclusão

Nesse trabalho foi implementado o algoritmo para índice invertido, porém a ordenação externa teve uma importância maior. Os resultados encontrados pelos experimentos e operações sobre arquivos comprovam o alto custo computacional quando um grande volume de dados excedem a memória principal e precisamos usar ordenação externa.

A eficiência da ordenação externa está fortemente dependentes do estado atual do hardware da máquina. Uma das maiores dificuldades deste trabalho consistiu em fazer uma intercalação eficiente dos arquivos e otimizar o tamanho da memória utilizado pelos dados do programa.

O trabalho possibilitou a prática da linguagem de programação C, a manipulação de arquivos e caracteres, o estudo de métodos de ordenação externa com relação a restrição da capacidade de memória principal da máquina.

6. Referências

[1] Cormen, T. (2001). Introduction to algorithms. Edição 2001. MIT press.

[2] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 2ª Edição, Editora Thomson, 2004.

[3] Slides preparados pela Prof. Olga Goussevskaia e disponibilizados para os alunos da disciplina de Algoritmos e Estruturas de Dados III.