TRABALHO I

Pesquisa e Ordenação de Dados

João Luis Almeida Santos

SUMÁRIO

O projeto implementado também pode ser acessado pelo repositório do GITHUB, através deste link.

Este relatório apresenta o desenvolvimento e a análise de uma solução voltada para a ordenação externa de números inteiros positivos, conforme proposto no Trabalho T1 da disciplina. O objetivo central é aplicar técnicas de manipulação de arquivos e otimização de uso de memória principal, simulando um cenário em que os dados excedem a capacidade de armazenamento interno e precisam ser processados em blocos. Apesar dos comentários estarem em português, a prática para nomear as funções e variáveis foram baseadas no inglês. O documento explora a lógica aplicada e discute as decisões de design algorítmico, identifica desafios enfrentados durante o desenvolvimento e realiza uma análise da complexidade do código. O código clona o arquivo arquivo original para um arquivo que podemos ler e alterar sem alterar o arquivo principal, INPUT. Após, há a ordenação dos dados, e salvamento dos resultados no arquivo SAIDA.TXT. Este relatório estuda as decisões críticas tomadas durante a realização do trabalho, as estratégias de manipulação de arquivos temporários, e análise da complexidade.

Aqui estão as bibliotecas utilizadas e as variáveis globais usadas no código:

```

// Configurado para o meu computador, alterar se for usar no Windows
#ifdef _WIN32
#define INPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\dados.txt"
#endif
#ifdef linux
#define INPUT "dados.txt"
#endif
#include <ctype.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

#define OUTPUT "saida.txt"
```

I. DESAFIOS E EVOLUÇÃO DO CÓDIGO

Na primeira versão do código, desenvolvi uma solução baseada no carregamento completo dos dados em memória principal usando um array, seguido da aplicação do *Insertion Sort*. Essa abordagem, disponível no GitHub[1], partia da premissa equivocada de que a ordenação poderia ser feita somente lendo os arquivos e colocando os números em um array. Na minha segunda tentativa, fiz um programa que realizava o processo de merge incorretamente. Para a versão final foi necessário mais pesquisa para entender como fazer apropriadamente o programa funcionar.

```
int main () {
    // Estou colocando 15 números
    int data_amount;
    printf("Insira a quantidade de elementos:\n");
    if (scanf("%d", &data_amount) != 1 || data_amount <= 0){
        printf("[ERRO] Entrada inválida\n");
        return 0;
    };
    int* list = malloc(data_amount * sizeof(int));
    if (list == NULL)
    {
        printf("[ERRO] Falha ao alocar memória.\n");
        return 1;
    }
    makeNumList(data_amount, list);

    printf("LISTA ORIGINAL:\n");
    printIntList(list, data_amount);
    insertionSort(list, data_amount);
    // Insere a lista em um arquivo, numeros separados via ";"
    intListOrderToFile(list, data_amount);
    free(list);

    return 0;
}
```

[1] https://github.com/lyszt/Trabalho—POD/blob/main/antigo.c

```
}
```

A função **externalSort(int chunk_size)** segue estes passos: contamos quantos números existem no arquivo, criamos arquivos temporários para cada bloco de tamanho **chunk_size**, usamos **qsort** para ordenar cada bloco, realizamos o merge dos blocos ordenados e, por fim, removemos os arquivos temporários.

Fizemos melhorias como contagem exata de elementos, criação dinâmica de arquivos temporários, ordenação interna eficiente com **qsort** e merge em tempo $O(n \log m)$, além de cuidar automaticamente da memória.

Na contagem dos números, a função count_numbers_in_file lê todo o arquivo, ignora caracteres não numéricos e conta quantos inteiros existem. Usamos **fscanf** para isso.

Para ordenar cada bloco, carregamos até **chunk_size** números na memória e aplicamos:

```
qsort(bloco, tamanho, sizeof(int), compare);
```

No merge, repetimos a busca pelo menor valor entre todos os blocos e escrevemos esse valor no arquivo de saída, até processar todos os números. Inicialmente fiz a escolha de utilizar o bubble sort, mas descobri depois que o qsort (implementação do Quicksort) era bem mais prático e agilizava a escrita do código, e é uma função que já existe por padrão no C. Bastou-se criar a função de compare para utilizar junto.

O programa começa contando quantos números existem no arquivo. Isso é crucial para determinar quantos blocos serão criados. A função **count_numbers_in_file** ignora caracteres inválidos e conta apenas dígitos, garantindo que espaços ou símbolos acidentais não quebrem o programa. Para arquivos muito grandes, essa contagem é feita sem carregar tudo na memória. O código lê o arquivo pedaço por pedaço, como mostra este trecho de código:

```c
// Function for reading data
void makeNumList(const int size, int* num_list) {
    FILE *f = fopen(INPUT, "r");

    // A função vai adicionando os caracteres dos números a uma string
    // até encontrar um caracter vazio, ponto e vírgula, ou o fim do arquivo

    if (f == NULL) {
        printf("[ERRO] Não foi possível abrir o arquivo, saindo.\n");
        return;
    }

    int i = 0;
    int j = 0;
    char number[16];
    char c;

    while ((c = fgetc(f)) != EOF && j < size) {
        if (c == ' ' || c == '\n') {
            if (i > 0) {
                number[i] = '\0';
                num_list[j++] = strtol(number, NULL, 10);
                i = 0;
            }
            continue;
        }

        if (c == ';') {
            if (i > 0) {
                number[i] = '\0';
                num_list[j++] = strtol(number, NULL, 10);
                i = 0;
            }
        } else if (i < 15) {
            number[i++] = c;
        }

        if (j >= size) {
            break;
        }
    }
```

```
    // Trata o último número, se necessário
    if (i > 0 && j < size) {
        number[i] = '\0';
        num_list[j++] = strtol(number, NULL, 10);
    }

    fclose(f);
}
```

Um desafio foi lidar com o último bloco, que pode ter menos números que os outros. O código ajusta automaticamente o tamanho desse bloco final usando cálculo modular:

```
int tamanho = (i == total_blocos - 1) ? total % chunk_size : chunk_size;
```

A etapa final usa uma estratégia de "menor elemento primeiro". Todos os blocos ordenados são abertos simultaneamente. O programa sempre pega o menor número disponível entre os primeiros elementos de cada bloco e escreve no arquivo final.

```

void externalSort(int chunk_size) {
    printf("\n=== INICIANDO ORDENACAO EXTERNA ===\n");
    printf("Tamanho dos blocos: %d elementos\n", chunk_size);

    // 1. Verificar quantidade total de números
    printf("\n[PASSO 1] Contando numeros no arquivo...\n");
    int total = count_numbers_in_file(INPUT);
    if(total == -1) {
        printf("Falha contando quantidade de números no arquivo.\n");
        return;
    }
    if(total <= 0) {
        printf("Erro ao ler arquivo de entrada\n");
        return;
    }
    printf("Total de numeros encontrados: %d\n", total);

    // 2. Calcular quantos arquivos temporários serão criados
```

```
    int num_arquivos = (total + chunk_size - 1) / chunk_size;
    printf("\n[PASSO 2] Criando %d arquivos temporarios\n", num_arquivos);
    char nome_arquivo[256];

    // 3. Ler, ordenar e salvar em arquivos temporários
    printf("\n[PASSO 3] Processando blocos:\n");
    FILE *entrada = fopen(INPUT, "r");
    for(int i = 0; i < num_arquivos; i++) {
   // Calcular tamanho do bloco
        int tamanho;
        if (i == num_arquivos - 1) {
            tamanho = total % chunk_size;
        } else {
            tamanho = chunk_size;
        }        printf("\n-> Bloco %d/%d (%d elementos)\n", i+1, num_arquivos, tamanho);

        // Ler números
        printf("   Lendo do arquivo original...\n");
        int *bloco = malloc(tamanho * sizeof(int));
        for(int j = 0; j < tamanho; j++) {
            fscanf(entrada, "%d", &bloco[j]);
            while(fgetc(entrada) == ';');
        }

        // Ordenar e salvar
        printf("   Ordenando...\n");
        qsort(bloco, tamanho, sizeof(int), compare);

        sprintf(nome_arquivo, PROCESS_N, i);
        printf("   Salvando no arquivo temporario: %s\n", nome_arquivo);
        FILE *temp = fopen(nome_arquivo, "w");
        for(int j = 0; j < tamanho; j++) fprintf(temp, "%d;", bloco[j]);
        fclose(temp);
        free(bloco);
    }
    fclose(entrada);

    // 4. Fazer merge dos arquivos temporários
    printf("\n[PASSO 4] Iniciando merge dos arquivos:\n");
    // Ponteiro duplo é um array de arquivos FILE
    // Isso facilita aqui pra baixo
```

```
    FILE **arquivos = malloc(num_arquivos * sizeof(FILE*));
    int *valores = malloc(num_arquivos * sizeof(int));

    // Abrir arquivos temporários
    printf("Abrindo arquivos para merge...\n");
    for(int i = 0; i < num_arquivos; i++) {
        sprintf(nome_arquivo, PROCESS_N, i);
        arquivos[i] = fopen(nome_arquivo, "r");
        fscanf(arquivos[i], "%d;", &valores[i]);
    }

    // Escrever resultado ordenado
    int elementos_processados = 0;
    FILE *saida = fopen(OUTPUT, "w");
    printf("\n[PASSO 5] Gerando arquivo final:\n");
    while(1) {
        // Encontrar menor valor
        int menor_valor = INT_MAX;
        int indice_menor = -1;

        for(int i = 0; i < num_arquivos; i++) {
            if(valores[i] < menor_valor) {
                menor_valor = valores[i];
                indice_menor = i;
            }
        }

        if(indice_menor == -1) break;

        // Escrever e atualizar
        fprintf(saida, "%d;", menor_valor);
        elementos_processados++;

        if(elementos_processados % 1000 == 0) {
            printf("-> Progresso: %d elementos ordenados\n", elementos_processados);
        }

        if(fscanf(arquivos[indice_menor], "%d;", &valores[indice_menor]) != 1) {
            printf("Arquivo %d esvaziado\n", indice_menor);
            valores[indice_menor] = INT_MAX;
        }
```

```
    }

    // 5. Limpeza final
    printf("\n[PASSO 6] Finalizando processo:\n");
    for(int i = 0; i < num_arquivos; i++) {
        fclose(arquivos[i]);
        sprintf(nome_arquivo, PROCESS_N, i);
        remove(nome_arquivo);
        printf("Removido arquivo temporario: %s\n", nome_arquivo);
    }
    free(arquivos);
    free(valores);
    fclose(saida);

    printf("\n=== PROCESSO CONCLUIDO ===\n");
    printf("Arquivo final gerado: %s\n", OUTPUT);
    printf("Total de elementos ordenados: %d\n\n", elementos_processados);
}
}
```

II. ANÁLISE DE COMPLEXIDADE

- **n**: Número total de elementos
- **k**: Tamanho do bloco (chunk_size)
- **m**: Número de blocos (m = n/k)

```
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);  // O(1)
}

void cloneFile(const char* origin, const char* destination) {
   O(M) onde M = tamanho do arquivo
}

int count_numbers_in_file(const char *filename) {
    O(M) onde M = tamanho do arquivo
}

```

```
void externalSort(int chunk_size) {
    // O(M + N log C + N²/C)
}

int main() {
    // O(1) + O(M + N log C + N²/C)
    // = O(M + N log C + N²/C)
}
}

```

$$\mathcal{O}\left(M + N \log C + \frac{N^2}{C}\right)$$

III. CONCLUSÃO

A partir do presente trabalho, é possível concluir que o algoritmo desenvolvido demonstrou com sucesso os princípios fundamentais da ordenação externa utilizando arquivos de texto. O código, quando executado, se apresenta como demonstrado na figura 1. A fase de sort performa mais rapidamente quando executado com mais chunks. E a de merge, ao contrário.

```
Quantidade de elementos por vez:
1
Sa├¡da final dispon├¡vel em [ SAIDA.TXT ]
```

Figura 1: Console na execução do código.