Universidade Federal de Mato Grosso
Campus Universitário do Araguaia
Instituto de Ciências Exatas e da Terra
Curso: Ciência da Computação

Estrutura de Dados I
Análise e Explicação do Sistema de Ordenação de Filmes

Discente: João Paulo Alves Campos
RGA: 202411722016
Professor : Dr Ivairton M. Santos

Relatório: Análise e Explicação do Sistema de Ordenação de Filmes

Introdução

Neste relatório, apresentarei uma análise detalhada do código de um sistema de ordenação de filmes implementado em linguagem C. Compreender a estrutura e o funcionamento deste código é fundamental para apreciar a elegância dos algoritmos de ordenação e sua aplicação prática em um contexto real de manipulação de dados.

Estrutura Geral do Código

Ao analisar o código fornecido, percebo que ele foi estruturado de forma modular e bem organizada, seguindo princípios de boas práticas de programação. Esta abordagem modular facilita a manutenção do código e permite uma compreensão mais clara de cada componente do sistema.

O sistema está dividido em vários arquivos, cada um com uma responsabilidade específica:

1. **Movie.h e movie.c**: Definem e implementam as estruturas de dados e funções relacionadas aos filmes. Considero esta separação essencial, pois isola a lógica de representação e manipulação dos dados de filmes do restante do sistema.

2. **sort.h e sort.c**: Contêm as declarações e implementações dos algoritmos de ordenação. Esta separação é importante para manter o código organizado e permitir a reutilização dos algoritmos em diferentes contextos.

3. **main.c**: Implementa a função principal do programa, integrando os componentes anteriores e fornecendo a interface com o usuário. Na minha análise, este arquivo serve como o ponto central que orquestra todas as funcionalidades do sistema.

Estruturas de Dados

No arquivo Movie.h, são definidas duas estruturas principais:

```
typedef struct {
    int movieId;
    char title[100];
    int year;
    char genre[50];
    float avgRating;
} Movie;

typedef struct {
    Movie *movies;
    int size;
    int capacity;
} MovieList;
```

A estrutura Movie representa um filme individual com seus atributos, enquanto MovieList implementa uma lista dinâmica de filmes. Esta abordagem é muito eficiente, pois a lista dinâmica permite que o programa trabalhe com um número variável de filmes sem desperdiçar memória.

Além disso, o sistema define uma enumeração para as opções de ordenação:

```
typedef enum {
    ASCENDING = 0,
    DESCENDING = 1
} SortOrder;
```

Esta enumeração torna o código mais legível e menos propenso a erros, pois substitui valores mágicos (0 e 1) por nomes significativos. Este tipo de abordagem melhora significativamente a manutenibilidade do código.

Fluxo de Execução

O fluxo de execução do programa segue uma sequência lógica:

1. Inicialização da lista de filmes
2. Carregamento dos dados de filmes e avaliações a partir de arquivos CSV
3. Apresentação de um menu interativo para o usuário

4. Aplicação do algoritmo de ordenação escolhido com base nos critérios selecionados
5. Exibição dos resultados ordenados
6. Repetição do processo ou encerramento do programa

Este fluxo é intuitivo e proporciona uma boa experiência ao usuário, permitindo múltiplas ordenações sem a necessidade de recarregar os dados.

Integração dos Componentes

A integração entre os diferentes componentes do sistema é realizada de forma elegante através de interfaces bem definidas. Por exemplo, os algoritmos de ordenação em sort.c recebem ponteiros para arrays de filmes e utilizam uma função de comparação que implementa diferentes critérios de ordenação.

Esta abordagem de design permite que os algoritmos de ordenação sejam genéricos o suficiente para trabalhar com qualquer critério de comparação, demonstrando um bom uso do princípio de separação de responsabilidades. Esta é uma característica fundamental de um código bem projetado.

Algoritmos de Ordenação

O sistema implementa três algoritmos clássicos de ordenação: Merge Sort, Insertion Sort e Quick Sort. Cada um desses algoritmos possui características distintas que os tornam mais adequados para diferentes cenários.

Merge Sort

O Merge Sort é implementado através das funções `mergeSort` e `fusao`. Analisando o código, percebo que ele segue fielmente o paradigma “dividir para conquistar”:

```
void mergeSort(Movie *arr, int esquerda, int direita, int criterio, int ordem) {
    if (esquerda < direita) {
        int meio = esquerda + (direita - esquerda) / 2;
        mergeSort(arr, esquerda, meio, criterio, ordem);
        mergeSort(arr, meio + 1, direita, criterio, ordem);
        fusao(arr, esquerda, meio, direita, criterio, ordem);
    }
}
```

A implementação do Merge Sort neste sistema é particularmente eficiente por várias razões:

1. Utiliza a fórmula esquerda + (direita - esquerda) / 2 para calcular o ponto médio, evitando possíveis overflows que poderiam ocorrer com a fórmula mais simples (esquerda + direita) / 2.
2. A função fusao aloca memória dinamicamente para os arrays temporários, liberando-a após o uso, o que demonstra uma preocupação com o gerenciamento adequado de recursos.
3. O algoritmo mantém a estabilidade, ou seja, elementos iguais mantêm sua ordem relativa original, o que é importante para ordenações por múltiplos critérios.

Insertion Sort

O Insertion Sort é implementado de forma concisa e eficiente:

```
void insertionSort(Movie *arr, int tamanho, int criterio, int ordem) {
    for (int i = 1; i < tamanho; i++) {
        Movie chave = arr[i];
        int j = i - 1;
        while (j >= 0 && compararFilmes(arr[j], chave, criterio, ordem) > 0) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = chave;
    }
}
```

Embora o Insertion Sort seja geralmente menos eficiente para grandes conjuntos de dados (com complexidade $O(n^2)$ no pior caso), ele possui vantagens significativas em determinados cenários:

1. É simples de implementar e entender
2. É eficiente para pequenos conjuntos de dados
3. Tem bom desempenho em arrays quase ordenados
4. Requer pouca memória adicional (ordenação in-place)

Quick Sort

O Quick Sort é implementado através das funções quickSort e particionar:

```
void quickSort(Movie *arr, int inicio, int fim, int criterio, int ordem) {
    if (inicio < fim) {
        int pi = particionar(arr, inicio, fim, criterio, ordem);
        quickSort(arr, inicio, pi - 1, criterio, ordem);
        quickSort(arr, pi + 1, fim, criterio, ordem);
    }
}
```

A implementação do Quick Sort neste sistema é bastante tradicional, utilizando o último elemento como pivô. Esta escolha simplifica o código, mas pode levar a um desempenho ruim em arrays já ordenados. No entanto, para a maioria dos casos práticos, o Quick Sort oferece um excelente desempenho médio, com complexidade O(n log n).

Função de Comparação

Um aspecto crucial dos algoritmos de ordenação é a função de comparação compararFilmes, que implementa diferentes critérios de ordenação:

```
int compararFilmes(Movie a, Movie b, int criterio, int ordem) {
    int resultado;

    if (criterio == 0) { // Título
        // Comparação de título insensível a maiúsculas/minúsculas
        char tituloA[100], tituloB[100];
        strcpy(tituloA, a.title);
        strcpy(tituloB, b.title);

        // Converte para minúsculas para comparação
        for (int i = 0; tituloA[i]; i++) {
            tituloA[i] = tolower(tituloA[i]);
        }
        for (int i = 0; tituloB[i]; i++) {
            tituloB[i] = tolower(tituloB[i]);
        }

        resultado = strcmp(tituloA, tituloB);
    } else if (criterio == 1) { // Ano
        resultado = a.year - b.year;
    } else { // Avaliação
        if (a.avgRating > b.avgRating) resultado = -1;
        else if (a.avgRating < b.avgRating) resultado = 1;
```

```
        else resultado = 0;

        // Para avaliação, o padrão é decrescente (maiores primeiro)
        // então invertemos lógica do ordem
        if (ordem == ASCENDING) resultado = -resultado;
        return resultado;
    }

    // Para título e ano, aplicamos ordem normalmente
    return (ordem == DESCENDING) ? -resultado : resultado;
}
```

Esta função é um excelente exemplo de código bem projetado por várias razões:

1. Implementa comparações específicas para cada tipo de critério
2. Trata a comparação de strings de forma insensível a maiúsculas/minúsculas
3. Lida corretamente com a inversão da ordem (crescente/decrescente)
4. Possui um caso especial para avaliações, onde a ordem natural é decrescente

```
MovieList* criarListaFilmes(int capacidadeInicial) {
    MovieList *lista = (MovieList*)malloc(sizeof(MovieList));
    if (!lista) {
        perror("Falha na alocação de memória");
        exit(EXIT_FAILURE);
    }
    lista->movies = (Movie*)malloc(capacidadeInicial * sizeof(Movie));
    if (!lista->movies) {
        perror("Falha na alocação de memória");
        free(lista);
        exit(EXIT_FAILURE);
    }
    lista->size = 0;
    lista->capacity = capacidadeInicial;
    return lista;
}
```

(filmes com maiores avaliações primeiro)

Manipulação de Filmes

A manipulação de filmes é implementada principalmente no arquivo movie.c, que contém funções para criar e gerenciar listas de filmes, carregar dados de arquivos CSV e exibir resultados.

Gerenciamento de Memória

O sistema utiliza alocação dinâmica de memória para gerenciar a lista de filmes:

Esta implementação é robusta por várias razões:

1. Verifica falhas de alocação de memória e trata os erros adequadamente
2. Inicializa todos os campos da estrutura
3. Libera recursos parcialmente alocados em caso de falha

A função adicionarFilme também demonstra boas práticas de gerenciamento de memória, expandindo dinamicamente a capacidade da lista quando necessário:

```
void adicionarFilme(MovieList *lista, Movie filme) {
    // Verifica se o filme tem dados válidos antes de adicionar
    if (!ehFilmeValido(&filme)) {
        return; // Descarta filmes inválidos
    }

    if (lista->size >= lista->capacity) {
        lista->capacity *= 2;
        Movie *temp = (Movie*)realloc(lista->movies, lista->capacity * sizeof(Movie));
        if (!temp) {
            perror("Falha na realocação de memoria");
            exit(EXIT_FAILURE);
        }
        lista->movies = temp;
    }
    lista->movies[lista->size++] = filme;
}
```

A estratégia de duplicar a capacidade quando necessário é um bom compromisso entre eficiência de memória e desempenho, pois reduz o número de realocações necessárias à medida que a lista cresce.

Carregamento de Dados

A função carregarFilmesDoCSV é responsável por carregar dados de filmes e avaliações a partir de arquivos CSV:

```
void carregarFilmesDoCSV(MovieList *lista, const char *arquivoFilmes, const char *arquivoAvaliacoes) {
    // ... (código omitido para brevidade)
}
```

Esta função é particularmente interessante por várias razões:

1. Processa dois arquivos CSV diferentes (filmes e avaliações)
2. Calcula médias de avaliações para cada filme
3. Utiliza arrays temporários para armazenar somas e contagens de avaliações
4. Implementa feedback visual do progresso durante o processamento
5. Lida com possíveis erros de formato nos arquivos CSV

A implementação desta função demonstra uma preocupação com a robustez e a experiência do usuário, características importantes em um sistema de qualidade.

A função analisarLinhaFilme merece destaque por sua complexidade:

```
int analisarLinhaFilme(const char* linha, Movie* filme) {
    // ... (código omitido para brevidade)
}
```

Esta função implementa um parser personalizado para lidar com as peculiaridades do formato CSV, incluindo campos entre aspas e extração do ano a partir do título do filme. Esta abordagem, embora mais complexa do que usar uma biblioteca de parsing CSV, oferece maior controle sobre o processo de extração de dados e permite lidar com casos específicos do formato dos dados.

Função Principal (main)

A função main no arquivo main.c é responsável por orquestrar todo o fluxo do programa:

```
int main() {
    printf("Iniciando o programa...\n");
    MovieList *lista = criarListaFilmes(1000);

    printf("Carregando filmes...\n");
    carregarFilmesDoCSV(lista, "movies.csv", "ratings.csv");
    if (lista->size == 0) {
        printf("Nenhum filme foi carregado. Verifique os arquivos CSV.\n");
        liberarListaFilmes(lista);
        return 1;
    }
    printf("Carregados %d filmes.\n", lista->size);

    // ... (código do loop principal omitido para brevidade)

    liberarListaFilmes(lista);
```

```
    printf("Programa encerrado.\n");
    return 0;
}
```

O loop principal do programa implementa um menu interativo que permite ao usuário escolher diferentes critérios, algoritmos e ordens de ordenação:

```
while (1) {
    exibirMenu();
    printf("Aguardando entrada do usuario...\n");
    if (scanf("%d %d %d", &criterio, &algoritmo, &ordem) != 3) {
        printf("Entrada invalida. Tente novamente.\n");
        while (getchar() != '\n'); // Limpa buffer
        continue;
    }

    // ... (validação e processamento das escolhas)

    // Criar cópia da lista para ordenação
    Movie *copiaFilmes = (Movie*)malloc(lista->size * sizeof(Movie));
    if (!copiaFilmes) {
        perror("Falha na alocacao de memoria");
        liberarListaFilmes(lista);
        return 1;
    }
    memcpy(copiaFilmes, lista->movies, lista->size * sizeof(Movie));

    // ... (aplicação do algoritmo de ordenação e exibição dos resultados)

    free(copiaFilmes);

    printf("\nDeseja continuar? (1 = Sim, 0 = Nao): ");
    int continuar;
    scanf("%d", &continuar);
    if (!continuar) break;
}
```

Esta implementação é particularmente inteligente por várias razões:

1. Cria uma cópia dos dados antes de aplicar a ordenação, preservando a lista original para futuras ordenações
2. Valida as entradas do usuário para evitar comportamentos inesperados
3. Mede o tempo de execução do algoritmo de ordenação, permitindo comparações de desempenho
4. Fornece feedback claro ao usuário sobre as operações realizadas

5. Implementa um loop que permite múltiplas ordenações sem reiniciar o programa

Considerações sobre Eficiência e Desempenho

Analisando o código como um todo, percebo várias decisões de implementação que afetam a eficiência e o desempenho do sistema:

1. **Algoritmos de ordenação**: A disponibilidade de três algoritmos diferentes permite escolher o mais adequado para cada cenário. O Merge Sort e o Quick Sort têm complexidade O(n log n) no caso médio, enquanto o Insertion Sort tem complexidade O(n²), mas pode ser mais eficiente para pequenos conjuntos de dados.

2. **Alocação de memória**: O sistema utiliza alocação dinâmica de memória com estratégia de crescimento exponencial (duplicação da capacidade), o que oferece um bom equilíbrio entre uso de memória e número de realocações.

3. **Processamento de CSV**: O parser personalizado para arquivos CSV pode não ser tão eficiente quanto bibliotecas especializadas, mas oferece maior controle sobre o processo de extração de dados.

4. **Cópia de dados para ordenação**: A criação de uma cópia dos dados antes da ordenação preserva a lista original, mas aumenta o uso de memória. Este é um compromisso razoável, pois permite múltiplas ordenações sem recarregar os dados.

Tempo de execução

A seguir a tabela contendo o tempo de execução de cada método de ordenação :

Critérios	**Insertion Sort**	**Merge Sort**	**Quick sort**
Titulo	1658.191(27min)	2.376s	1.283s
Ano	308.7180s	0.264s	5.421s
nota	85.348s	0.263s	5.993s

Segue o ranking de melhor eficiência em ordenação:

1. **Merge Sort** (n log n) - Alta eficiencia
2. **Quick sort** (n^2 no pior caso) - Media eficiencia
3. **Insertion Sort** (n^2) - Baixa eficiência

*Todos os testes foram feitos em ordem Decrescente".

Conclusão

Após uma análise minuciosa do código do sistema de ordenação de filmes, é possível afirmar que ele constitui um exemplo notável de como algoritmos de ordenação podem ser aplicados de maneira prática e eficiente em um contexto real. A estrutura modular do sistema, aliada à escolha cuidadosa de algoritmos como Merge Sort, Quick Sort e Insertion Sort, reflete um equilíbrio bem-sucedido entre funcionalidade, desempenho e manutenibilidade. Argumento que a decisão de implementar esses três algoritmos, cada um com suas particularidades, demonstra uma preocupação em atender a diferentes necessidades: o Merge Sort se destaca pela estabilidade e eficiência em grandes conjuntos de dados, o Quick Sort oferece um desempenho médio robusto, enquanto o Insertion Sort brilha em cenários com listas menores ou quase ordenadas. Essa diversidade fortalece o sistema, permitindo flexibilidade e adaptação a variados volumes e tipos de entrada.

Além disso, a gestão dinâmica de memória, com estratégias como a duplicação da capacidade da lista, reforça a eficiência do programa ao minimizar realocações frequentes, embora implique um custo adicional de memória que se justifica pela preservação da lista original para múltiplas ordenações. Tal abordagem, somada ao carregamento robusto de dados via CSV e à interface interativa amigável, evidencia um design que prioriza tanto a experiência do usuário quanto a integridade dos dados. Os tempos de execução apresentados — com o Merge Sort consistentemente mais rápido, seguido pelo Quick Sort e, por fim, pelo Insertion Sort — corroboram a superioridade teórica da complexidade O(n log n) em cenários práticos, embora o Insertion Sort ainda tenha seu mérito em contextos específicos.