Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Ordenação de Objetos Móveis

BCC202 - Estruturas de Dados I

Bruno Alves Braga
Kailainy do Patrocínio
Kézia Alves Brito
Professor: Pedro Silva

Ouro Preto
19 de fevereiro de 2023

Sumário

1 Introdução

Para este trabalho, é necessário entregar o código implementado na linguagem de programação C e o relatório em questão, referente ao que foi desenvolvido. O algoritmo programado consiste na ordenação de objetos móveis com base na trajetória.

1.1 Especificações do problema

O objetivo deste trabalho é codificar uma solução que, através de um conjunto de trajetórias, calcule o deslocamento e a distância percorrida por n objetos entre m pontos, além de ordenar esses objetos com base em 3 (três) critérios: ordem decrescente da distância percorrida; crescente com base no deslocamento e ordem crescente do identificador/nome do objeto móvel.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: *Visual Studio Code + LiveShare*.
- Linguagem utilizada: C.
- Compilador utilizado: GCC *version* 12.2.1 20230201.
- Ambiente de desenvolvimento da documentação: *Visual Studio Code* + LaTeX *Workshop*.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- Valgrind: ferramenta de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: Intel Core i5.
- Memória RAM: 8 GB.
- Sistema Operacional: Manjaro Linux.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-c*: para compilar o arquivo sem vincular os arquivos do tipo objeto.
- *-o*: para vincular os arquivos do tipo objeto.
- *-Wall*: para mostrar todos os possíveis *warnings* do código.
- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.
- *-lm*: para *linkar* a biblioteca *math*.

Para a execução do programa, basta digitar:

```
./exe numero_de_objetos numero_de_pontos
./exe < arquivo_de_entrada
```

2 Implementação

Em *ordenacao.h* foram criados dois Tipos Abstratos de Dados - TADs, referentes aos objetos e aos pontos, necessários para a construção do programa.

```
typedef struct {
    float x;
    float y;
} Ponto;

typedef struct {
    char nome[5];
    Ponto *trajetoria;
    float distancia;
    float deslocamento;
} Objeto;
```

2.1 *alocaObjetos*

Recebe a quantidade total de objetos e de pontos e aloca, dinamicamente, um vetor de objetos, alocando também um vetor de pontos (trajetória) dentro de cada objeto, através da função *alocaPontos*. Essa função retorna um vetor de TAD Objeto, que vai ser atribuído a outro vetor de mesmo tipo na *main*.

```
Objeto * alocaObjetos(int totalObjetos, int totalPontos) {

    Objeto *objetos = (Objeto*) malloc(totalObjetos * sizeof(Objeto));

    alocaPontos(objetos, totalObjetos, totalPontos);

    return objetos;
}

void alocaPontos(Objeto *objetos, int totalObjetos, int totalPontos) {

    for(int i = 0; i < totalObjetos; i++) {
        objetos[i].trajetoria = (Ponto*) malloc(totalPontos * sizeof(Ponto));
    }
}
```

2.2 *leObjetos*

Recebe um vetor de objetos alocado e lê o nome de cada objeto, assim como os pontos de sua trajétoria.

```
void leObjetos(Objeto *objetos, int totalObjetos, int totalPontos) {

    for(int i = 0; i < totalObjetos; i++) {
        scanf("%s", objetos[i].nome);

        for(int j = 0; j < totalPontos; j++) {
            scanf("%f %f", &objetos[i].trajetoria[j].x, &objetos[i].trajetoria
                [j].y);
        }
    }
}
```

2.3 *teoremaPitagoras*

Para facilitar os cálculos, criamos uma função que realiza o Teorema de Pitágoras.

```
float teoremaPitagoras(int x1, int y1, int x2, int y2) {

    float resultado = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));

    return resultado;
}
```

2.4 *calculaDistancia*

Calcula a distância total percorrida por um objeto dada a quantidade de pontos e as coordenadas de cada um. O cálculo é realizado a partir do somatório das distâncias entre cada par de pontos, o valor final sendo atribuído à variável *distancia* da *struct* de objetos.

```
void calculaDistancia(Objeto *objetos, int totalObjetos, int totalPontos) {

    float distancia = 0.0;

    for(int i = 0; i < totalObjetos; i++) {
        for(int j = 0; j < totalPontos - 1; j++) {
            distancia += teoremaPitagoras(objetos[i].trajetoria[j].x,
                objetos[i].trajetoria[j].y,
                objetos[i].trajetoria[j + 1].x,
                objetos[i].trajetoria[j + 1].y);
        }

        objetos[i].distancia = distancia;
        distancia = 0.0;
    }
}
```

2.5 *calculaDeslocamento*

Calcula o deslocamento final de um objeto pelo Teorema de Pitágoras, realizado com as coordenadas do primeiro e último pontos por ele percorridos. Atribui o valor encontrado à variável *deslocamento* dentro do TAD de *Objeto*.

```
void calculaDeslocamento(Objeto *objetos, int totalObjetos, int totalPontos) {

    for(int i = 0; i < totalObjetos; i++) {
        objetos[i].deslocamento = teoremaPitagoras(objetos[i].trajetoria[0].x,
            objetos[i].trajetoria[0].y,
            objetos[i].trajetoria[totalPontos - 1].x,
            objetos[i].trajetoria[totalPontos - 1].y);
    }
}
```

2.6 *ordenaObjetos*

Utiliza o método de ordenação *Shell Sort* para ordenar um vetor de objetos em ordem decrescente de acordo com a distância total percorrida. Caso os valores de distância sejam iguais, arranja de forma crescente de acordo com os deslocamentos e, caso estes também sejam iguais, ordena alfabeticamente.

```
void ordenaObjetos(Objeto *objetos, int totalObjetos) {

    int h, j;
    Objeto auxiliar;

```

```
    for(h = 1; h < totalObjetos;) {
        h = 3 * h + 1;
    }

    do {
        h = (h - 1) / 3;

        for(int i = h; i < totalObjetos; i++) {
            auxiliar = objetos[i];
            j = i;

            while(comparaObjetos(&objetos[j - h], &auxiliar) == 1) {
                objetos[j] = objetos[j - h];
                j = j - h;

                if(j < h) {
                    break;
                }
            }

            objetos[j] = auxiliar;
        }
    } while(h != 1);
}
```

2.7 *comparaObjetos*

Auxiliando a função de ordenação, a *comparaObjetos* é responsável por retornar se dois valores devem (retorna 1) ou não (retorna -1) ser trocados, de acordo com os critérios pré-estabelecidos.

Os campos que armazenam a distância e o deslocamento de cada objeto são do tipo *float*, ou seja, são números de ponto flutuante. A precisão de comparação entre dois números desse tipo, em C, é de 0,01, portanto, se o módulo da diferença entre dois *floats* for menor ou igual a esse valor, os dois são iguais. Caso contrário, se A - B $>$ 0,01, A $>$ B; senão, B - A $>$ 0,01 e B $>$ A.

```
int comparaObjetos(Objeto *objeto1, Objeto *objeto2) {

    if((fabsf(objeto1->distancia - objeto2->distancia)) > 0.01) {

        if((objeto1->distancia - objeto2->distancia) > 0.01)
            return -1;

        else if((objeto2->distancia - objeto1->distancia) > 0.01)
            return 1;
    }

    else {

        if((fabsf(objeto1->deslocamento - objeto2->deslocamento)) > 0.01) {

            if((objeto1->deslocamento - objeto2->deslocamento) > 0.01)
                return 1;

            else if((objeto2->deslocamento - objeto1->deslocamento) > 0.01)
                return -1;
        }
```

Para ordenar alfabeticamente, a função utiliza a *strcmp*, que retorna a soma dos valores de duas *strings*. Se a soma for $= 0$, as *strings* são iguais; senão, se a soma for < 0, a segunda *string* é maior; senão, a soma é > 0, ou seja, a primeira *string* é maior. A maior *string* é a última em ordem alfabética.

```
        else {

            if(strcmp(objeto1->nome, objeto2->nome) > 0)
                return 1;

            else
                return 0;
        }
    }

    return 0;
}
```

2.8 *imprimeVetor*

Imprime o vetor de objetos ordenado, mostrando nome, distância e deslocamento de cada objeto.

```
void imprimeVetor(Objeto *objetos, int totalObjetos) {

    for(int i = 0; i < totalObjetos; i++) {
        printf("%s %.2f %.2f\n", objetos[i].nome, objetos[i].distancia,
            objetos[i].deslocamento);
    }
}
```

2.9 *desalocaObjetos*

Libera um vetor de pontos (trajetória) a partir da função *desalocaPontos* e, em seguida, libera um vetor de objetos.

```
void desalocaObjetos(Objeto *objetos, int totalObjetos) {

    desalocaPontos(objetos, totalObjetos);

    free(objetos);
}

void desalocaPontos(Objeto *objetos, int totalObjetos) {

    for(int i = 0; i < totalObjetos; i++) {
        free(objetos[i].trajetoria);
    }
}
```

3 Impressões gerais

O algoritmo desenvolvido neste trabalho é simples e objetivo. Sendo um algoritmo de análise exploratória de dados, a ideia principal é receber um conjunto de dados e, a partir deste, resumir as características principais. Muitas vezes estas características podem ser demonstradas em métodos visuais, como um gráfico.

De forma geral, a maior parte das funções implementadas são frequentemente utilizadas em outros trabalhos e atividades práticas, o que foi bem agradável.

Como essa foi a primeira vez que implementamos um método de ordenação eficiente em um programa mais elaborado, no entanto, conseguimos adquirir e fixar nossos conhecimentos sobre a ordenação de um conjunto de objetos, especialmente com a utilização do *Shell Sort*.

4 Análise

O método de ordenação utilizado foi o *Shell Sort*, que é basicamente uma extensão do *Insertion Sort*. Com base nos resultados obtidos, do ponto de vista do grupo, o *Shell Sort* é um dos mais eficientes métodos de ordenação, apesar de sua complexidade ser incerta. Além de eficiente, o *Shell Sort* é *in situ* e mais fácil de implementar em comparação a outros métodos, como o *Merge Sort*, por exemplo.

Além do método de ordenação, não há nenhum detalhe técnico que impacte muito no desempenho do algoritmo.

5 Conclusão

O processo de implementação foi rápido e descomplicado, sendo realizado em algumas horas. Não houve muita dificuldade em entender o que deveria ser feito no trabalho, o que facilitou o início do desenvolvimento. Uma das maiores dificuldades foi escolher e implementar o melhor método de ordenação para o caso. Tentamos, inicialmente, implementar o *Merge Sort*, mas logo encontramos dificuldade. Optamos, então, por implementar o *Shell Sort*, que funcionou perfeitamente.

Outra dificuldade encontrada durante o processo do trabalho foi a comparação entre números de ponto flutuante para ordenar o vetor com base na distância e deslocamento, uma vez que, na linguagem C, há uma margem de erro na precisão das casas decimais.