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

Trabalho Prático II

BCC202 - Estruturas de Dados I

Vitor Oliveira Diniz
Maria Luiza Aragão
Jéssica Machado
Professor: Pedro Silva

Ouro Preto
19 de fevereiro de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Neste trabalho foi necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é uma ordenação de objetos móveis com base em sua trajetória.

A codificação foi feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais.

1.1 Especificações do problema

No contexto de objetos móveis, as trajetórias são um conjunto de pontos das posições ocupadas pelo objeto durante seu movimento. Uma trajetória pode ser representada em função de uma sequência de dois ou mais pontos com coordenadas (x,y) distintas. Assim, devemos, com base em uma lista de objetos móveis e o total de pontos de cada trajetória, ambos quantativamente desconhecidos, ordenar a lista de saída nos baseando nas seguintes regras:

- decrescentemente, com base na distância percorrida;
- crescentemente, com base no o deslocamento e nome, respectivamente, caso haja empate no item anterior.

1.2 Considerações Iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Visual Studio Code LaTeXWorkshop.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *CLANG*: ferramentas de análise estática do código.
- *Valgrind*: ferramentas 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: Ryzen 7-5800H.
- Memória RAM: 16 Gb.
- Sistema Operacional: Arch Linux x86_64.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc tp.o ordenacao.o compare_double.o -o exe -g -Wall -lm
gcc tp.c -c -g -lm
gcc ordenacao.c -c -g -lm -Wall
gcc compare_double.c -c -g -lm -Wall
```

Usou-se para a compilação as seguintes opções:

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

- *-Wall*: para mostrar todos os possível *warnings* do código.

- *-c*: Para compilar o arquivo sem linkar os arquivos para obtermos um arquivo do tipo objeto.

- *-lm*: Para fazer o link da biblioteca math.h

- *-o*: Para indicar o nome do arquivo de saida.

Para a execução do programa basta digitar:

```
./exe < caminho_até_o_arquivo_de_entrada
```

Onde "caminho-até-o-arquivo-de-entrada" pode ser: "1.in" para realizar o primeiro caso de teste e "2.in" para realizar o segundo.

2 Desenvolvimento

Seguindo as boas práticas de programação, implementamos um tipo abstrato de dados (TAD) para a representação do nosso problema. De acordo com o pedido, e para uma melhor organização, o nosso código foi modularizado em cinco arquivos, tp.c ordenacao.h, ordenacao.c, compare_doubles.c e compare_doubles.h em que o arquivo tp.c deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo ordenacao.h.

2.1 TAD

Para começar a resolução do problema proposto, decidimos criar duas structs, que representariam o nosso objeto e seriam nosso TAD. O primeiro, struct Ponto, que nos informaria, com base no plano cartesiano, as coordenadas do Objeto, nossa segunda struct, que contém informações necessárias para distinguir um objeto do outro, seja sua trajetória, distância e ID.

```
1
2  typedef struct{
3      int x;
4      int y;
5  }Ponto;
6
7  typedef struct{
8      char ID[5];
9      Ponto *pontos;
10     int npontos;
11     double distancia;
12     double deslocamento;
13 }Objeto;
```

Código 1: TAD's representando Ponto e Objeto, respectivamente

2.2 Funções

A seguir entraremos em detalhe sobre as principais funções utilizadas no programa.

2.2.1 alocaPontos

Criamos essa função para alocar dinâmicamente, vários pontos, baseado em um valor digitado pelo usuário.

```
1
2  Ponto* alocaPontos (int npontos){
3
4      Ponto *pontos = (Ponto*)malloc(npontos * sizeof(Ponto));
5      return pontos;
6  }
```

Código 2: Função alocaPontos

2.2.2 alocaObjetos

Essa função aloca dinâmicamente um número X de objetos. Em seguida, com a ajuda de uma estrutura de repetição, adicionamos em cada um dos objetos o número de pontos, alocamos os pontos e definimos o valor inicial do deslocamento e da distância.

```
1
2  Objeto* alocaObjetos (int npontos, int nobj){
3
4      Objeto* objetos = (Objeto*)malloc(nobj * sizeof(Objeto));
5
6      for (int i = 0; i < nobj; i++){
```

```
7
8           objetos[i].npontos = npontos;
9           objetos[i].pontos = alocaPontos(npontos);
10
11          objetos[i].deslocamento = 0;
12          objetos[i].distancia = 0;
13
14      }
15
16      return objetos;
17  }
```

Código 3: Função alocaObjetos

2.2.3 desalocaPontos e desalocaObjetos

Esta função foi implementada com o objetivo de liberar o espaço de memória que alocamos para o ponto e o objeto.

```
1   void desalocaPontos (Ponto *pontos){
2       free(pontos);
3   }
4
5   void desalocaObjetos(Objeto **lista, int nobj){
6
7       for(int i = 0; i < nobj; i++){
8           desalocaPontos((*lista)[i].pontos);
9           printf("I: %d\n", i);
10      }
11      free(*lista);
12  }
```

Código 4: Funções desalocaPontos e desalocaObjetos

2.2.4 lerPontos e lerObjetos

Na função lerPontos, utilizamos uma estrutura de repetição que se repete até o número de pontos (npontos), váriavel passada por parâmetro, e, em cada um dos pontos, pegamos o valor de X e Y.
Já na função lerObjetos, a extrutura de repetição aconteceria até o número de objetos (nobj), também passada por parâmetro, e com o scanf pegariamos o ID do objeto e o valor de seus pontos, utilizando a função lerPontos

```
1   void lerPontos(Ponto* pontos, int npontos){
2       for (int i = 0; i < npontos; i++){
3           scanf("%d", &pontos[i].x);
4           scanf("%d", &pontos[i].y);
5       }
6   }
7
8
9   void lerObjetos(Objeto *lista, int nobj){
10      for(int i = 0; i < nobj; i++){
11          scanf("%s", lista[i].ID);
12          lerPontos(lista[i].pontos, lista->npontos);
13      }
14
15  }
```

Código 5: Funções lerPontos e lerObjetos

2.2.5 calcularDistancia e calcularDeslocamento

Para o cálculo dessas funções, utilizamos as fórmulas, já conhecidas, da distância e do deslocamento, e as funções da biblioteca math.h, que facilitou muito nossas contas.

```
1
2  double calcularDistancia (Objeto *objeto){
3      double distancia = 0;
4      for(int i = 0; i < objeto->npontos - 1; i++){
5          distancia += sqrt(pow((objeto->pontos[i+1].x - objeto->pontos[i].x),
               2) + pow((objeto->pontos[i+1].y - objeto->pontos[i].y), 2));
6      }
7      return distancia ;
8  }
9
10 double calcularDeslocamento (Objeto *objeto){
11     return sqrt(pow((objeto->pontos[objeto->npontos-1].x - objeto->pontos[0].x
           ), 2)
12 + pow((objeto->pontos[objeto->npontos-1].y - objeto->pontos[0].y), 2));
13 }
```

Código 6: Funções calcularDistancia e calcularDeslocamento

2.2.6 realizaCalculos

Utilizando uma estrutura de repetição, mudamos os valores da distância e do deslocamento de cada um dos objetos utilizando as funções calcularDistancia e calcularDeslocamento, anteriormente explicadas.

```
1  void realizaCalculos(Objeto *objetos, int nobj){
2      for (int i = 0; i < nobj; i++)
3      {
4          objetos[i].distancia = calcularDistancia(&objetos[i]);
5          objetos[i].deslocamento = calcularDeslocamento(&objetos[i]);
6      }
7
8  }
```

Código 7: Função realizaCalculos

2.2.7 comparaObjeto

Função necessária para fazer a comparação necessária de nossos TADs, seguindo as regras definidas no trabalho. Optamos por fazer uma função separada para que o código ficasse mais legível e modularizado. As funções utilizadas nessa função estão presentes no "compare_double.c".

```
1
2  int comparaObjeto(Objeto *objeto1, Objeto *objeto2){
3
4      if (definitelyGreaterThan(objeto1->distancia, objeto2->distancia)){
5          return 0;
6
7      } else if (definitelyLessThan(objeto1->distancia, objeto2->distancia)){
8          return 1;
9      } else {
10
11         if (definitelyGreaterThan(objeto1->deslocamento, objeto2->deslocamento
               )) {
12             return 1;
13
14         } else if(definitelyLessThan(objeto1->deslocamento, objeto2->
               deslocamento)){
15             return 0;
16         } else{
```

```
            if(strcmp(objeto1->ID, objeto2->ID) > 0){
                return 1;
            } else {
                return 0;
            }
        }

    }
}
```

Código 8: Função comparaObjetos

2.2.8 shellSort

Para ordenação, optamos por utilizar o método shellSort.

```

void shellSort(Objeto *objetos, int n) {
    int h = 1;
    Objeto aux;
    while( h < n)
        h = 3 * h + 1;

    do{

        h = (h - 1)/3;

        for ( int i = h; i < n;i++){
            aux = objetos[i];
            int j = i;
            while(comparaObjeto(&objetos[j - h], &aux)){
                objetos[j] = objetos[j-h];
                j = j - h;
                if ( j < h){
                    break;
                }
            }
            objetos[j] = aux;

        }
    } while ( h != 1);
}
```

Código 9: Função shellSort

2.2.9 mergeSort

Implementamos também a função mergeSorte, que faz uma chamada recursiva dela mesma no passo da divisão, para depois chamarmos a merge no passo da conquista e fazermos a ordenação.

```

void mergesort(Objeto *v, int l, int r, int npontos){
    if (l < r){
        int m = (l + r)/2;
        mergesort(v, l, m, npontos);
        mergesort(v, m + 1, r, npontos);
        merge(v, l, m, r, npontos);

    }
}
```

Código 10: Função mergeSort

2.2.10 merge

Implementamos a função merge, que será responsável pela parte da conquista, em que após a divisão irá ordenar os subvetores em um maior, até que nosso vetor esteja completamente ordenado.

```
1
2       void merge(Objeto *v, int l, int m, int r, int npontos){
3
4       int size_l = (m - l + 1);
5       int size_r = (r - m);
6
7       Objeto *vet_l = malloc( size_l * sizeof(Objeto));
8       Objeto *vet_r = malloc( size_r * sizeof(Objeto));
9
10
11      for (int i = 0; i < size_l; i++)
12          vet_l[i] = v[i + l];
13
14
15      for (int j = 0; j < size_r; j++)
16          vet_r[j] = v[m + j + 1];
17
18
19
20      int i = 0;
21      int j = 0;
22
23      for (int k = l; k <= r; k++){
24
25          if (i == size_l)
26              v[k] = vet_r[j++];
27
28          else if (j == size_r)
29              v[k] = vet_l[i++];
30
31          else if (comparaObjeto(&vet_l[i], &vet_r[j]))
32              v[k] = vet_r[j++];
33
34          else
35              v[k] = vet_l[i++];
36      }
37
38      free(vet_l);
39      free(vet_r);
40
41  }
```

Código 11: Função merge

2.3 main

Na função main, além de utilizarmos o scanf para obter o nobj e npontos, invocamos as funções necessárias para a realização dos procedimentos, sendo eles: a alocação dos objetos, a leitura deles, os cálculos necessários, a ordenação da lista, sua impressão e por último, desalocação.

```
1
2  #include <stdio.h>
3  #include "ordenacao.h"
4
5  int main(void){
6
7      Objeto *lista;
8      int npontos, nobj;
9
10     scanf("%d%d", &nobj, &npontos);
11
12     lista = alocaObjetos(npontos, nobj);
13     lerObjetos(lista, nobj);
14
15     realizaCalculos(lista, nobj);
16     //mergesort(lista, 0, npontos - 1, npontos);
17     shellSort(lista, nobj);
18     imprime(lista, nobj);
19     desalocaObjetos(&lista, nobj);
20
21     return 0;
22 }
```

Código 12: função main

3 Impressões Gerais

Primeiramente além das funções necessárias passadas no PDF, pensamos em quais funções a mais deveriamos implementar. Com as funções já pré definidas foi muito mais fácil construir a lógica para o desenvolvimento modular do código. O nosso grupo então se reuniu e pensou coletivamente sobre a ordem de execução das funções e suas utilidades.Outro conhecimento adquirido e posto em prática foi o uso de diferentes métodos de ordenação, para a solução do nosso problema. Inicialmente, obtivemos um pouco de dificuldade na implementação do MergeSort e acabamos optando pela implementação do ShellSort, porém depois com um pouco de dificuldade conseguimos resolver o problema e optamos por deixar os dois métodos no trabalho. Também tivemos um leve problema com a comparação de números de pontos flututantes, em que foi necessário uma pequena pausa no trabalho para que pudessemos pensar em uma solução. Foi ai que surgiu o compare_doubles.h. Assim houve também o desenvolvimento de um código bem modularizado, com uma excelente ajuda das instruções contidas no documento que nos foi disponibilizado como exemplo.

4 Análise

Após o desenvolvimento do programa, a primeira análise feita foi atráves dos casos de teste disponibilizados na página do trabalho prático do run.codes, com simples exemplos de entrada e saída, executamos o programa com um dos exemplos de entrada e assim, foi possível fazer uma simples análise se o programa se comportava corretamente. Obtivemos algumas dificuldades que serão abordadas na conclusão. Após a implementação de outro método de ordenação, visto que o primeiro não foi eficiente, as próximas realizações de testes apresentaram resultados iguais ao exemplo de saída disponibilizado. Depois dos testes iniciais para verificar um funcionamento inicial do programa, utilizamos o valgrind, uma ferramenta de análise dinâmica de código para conferir se há algum memory leak ou warning referente a manipulação de memória.

5 Conclusão

Com este trabalho ampliamos os nossos conhecimentos referente aos métodos de ordenação, em especial o shellSort e mergeSort, e como aplicá-lo para encontrar a solução de um problema. Como uma dificuldade inicial, tivemos a implementação do mergeSort que não gerou os resultados esperados, além de vários erros quando rodamos o valgrind. Esse erros eram devidos a liberação errônea do vetor auxiliar de objetos utilizados na função merge. Pois como faziamos uma atribuição de ponteiros o conteúdo dele não era copiado, apenas o ponteiro e ao realizar a liberação dele no final da função, era liberado o conteudo dos pontos causando um erro de double free e leitura inválida. Após um certo tempo, desistimos e utilizamos o shellSort. Porém após algum tempo pensando no problema, descobrimos nosso erro e decidimos reimplementar o mergeSort.