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

Trabalho Prático II (TP II)

BCC202 – Estruturas de Dados I

Bruno Soares Veríssimo

Professor: Pedro Silva

Ouro Preto
14 de dezembro de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é o caixeiro viajante.

A codificação deve ser feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais. Além disso, deve-se usar um dos padrões: ANSI C 89 ou ANSI C 99.

1.1 Especificações do problema

O Trabalho Prático II, propõe a resolução do Problema do Caixeiro Viajante. Neste cenário, um caixeiro deve visitar um conjunto de cidades distintas, começando e terminando na primeira cidade. O objetivo é determinar a trajetória que resulta na menor distância total de viagem. O trabalho exige a implementação de um algoritmo que utilize recursividade e listas de adjacências para encontrar o menor caminho possível passando por todas as cidades, partindo da cidade inicial.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

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: Intel® Core™ i5-10300H.
- Memória RAM: 24Gb.
- Sistema Operacional: Windows 10.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc main.c grafo.c -o exe -std=c99 -Wall -pg -g
```

Usou-se para a compilação as seguintes opções:

- *-std=99*: para usar-se o padrão ANSI C 99, conforme exigido.
- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.
- *-Wall*: para mostrar todos os possível *warnings* do código.
- *-pg*: para gerar o arquivo para fazer-se o profiling para identificar gargalos no programa.

[1] Visual Studio Code está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

Para a execução do programa basta digitar:

```
./exe caminho_até_o_arquivo_de_entrada opcao
```

Onde "opcao" pode ser: "p" para fazer a, "d" para fazer b e "n" para fazer c.

2 Desenvolvimento

O Trabalho Prático II é um trabalho muito parecido com o Trabalho Prático I. Ambos buscam a solução do problema do caixeiro viajante, Todavia dessa vez além de calcular o melhor caminho, deve-se usar listas de adjacências para representar os vizinhos de cada cidade, essas listas devem ser implementadas utilizando listas encadeadas, devem ser ordenadas e usadas para o cálculo do melhor caminho e da menor distância.

2.1 Criando as estruturas necessárias

Primeiramente foi necessário criar as estruturas No, que possuí o destino do nó, o peso da aresta que conecta o nó ao seu destino e o ponteiro para o próximo nó na lista, a estrutura ListaAdj, que possuí um ponteiro para a cabeça da lista de adjacência e a estrutura GrafoPonderado, que possuí o número de cidades no grafo e o vetor de listas de adjacência para cada cidade. Além disso, foi inicializado as funções alocarGrafo, desalocarGrafo, leGrafo, encontraCaminho, imprimeCaminho, ordenaLista, imprimeOrdenado e obterDistancia em grafo.h.

```
1  #ifndef GRAFO_H
2  #define GRAFO_H
3
4  // Defini  o da estrutura No, representando um n  em uma lista de
       adjac ncia.
5  typedef struct No {
6      int destino;        // Armazena o destino do n  (v rtice adjacente).
7      int peso;           // Armazena o peso da aresta que conecta o n  ao seu
           destino.
8      struct No* proximo;  // Ponteiro para o pr ximo n  na lista.
9  } No;
10
11 // Defini  o da estrutura ListaAdj, representando uma lista de adjac ncia
       para um v rtice em um grafo ponderado.
12 typedef struct ListaAdj {
13     No* cabeca;  // Ponteiro para a cabe a da lista de adjac ncia.
14 } ListaAdj;
15
16 // Defini  o da estrutura GrafoPonderado, representando um grafo ponderado.
17 typedef struct GrafoPonderado {
18     int numCidades;         // N mero de cidades no grafo.
19     ListaAdj* listaAdjacencia;  // Vetor de listas de adjac ncia para cada
           cidade.
20 } GrafoPonderado;
21
22 // Prot tipos das fun  es definidas no arquivo "grafo.c".
23 GrafoPonderado* alocarGrafo(int numCidades);
24 void desalocarGrafo(GrafoPonderado* grafo);
25 void leGrafo(GrafoPonderado* grafo);
26 void encontraCaminho(GrafoPonderado* grafo, int* melhorCaminho);
27 void imprimeCaminho(GrafoPonderado* grafo, int* caminho, int distancia);
28 void ordenaLista(GrafoPonderado* grafo);
29 void imprimeOrdenado(GrafoPonderado* grafo);
30 int obterDistancia(GrafoPonderado* grafo, int cidadeOrigem, int cidadeDestino)
       ;
31
```

```
32 #endif
```

Código 1: grafo.h

2.2 Resolução do problema

Após a criação das estruturas é necessário aloca-las. Para isso, a função criarNo aloca memória para um novo nó de grafo, atribui valores de destino e peso ao nó e inicializa o próximo nó como NULL e retorna o nó criado. A função criarListaAdj aloca memória para uma nova lista de adjacência, inicializa a cabeça da lista como NULL e retorna a lista de adjacência criada. E a função alocarGrafo aloca memória para um novo grafo ponderado, inicializa o número de cidades e aloca memória para a lista de adjacência e retorna o grafo alocado.

A função desalocarGrafo libera a memória de todos os nós e listas de adjacência, e do próprio grafo e a função leGrafo lê do input as informações das arestas do grafo e as armazena nas listas de adjacência correspondentes.

A função obterDistancia foi projetada para encontrar a distância ou o peso de uma aresta entre duas cidades específicas em um grafo ponderado. A função começa localizando a lista de adjacência da cidade de origem. Ela faz isso acessando grafo->listaAdjacencia[cidadeOrigem].cabeca. Este passo é crucial porque a lista de adjacência da cidade de origem contém todos os nós (ou arestas) que se conectam a outras cidades a partir da cidade de origem. A função então entra em um loop while que percorre a lista de adjacência. O loop continua enquanto o ponteiro para o nó atual (atual) não é NULL, o que significaria o fim da lista. Dentro do loop, a função verifica se o destino do nó atual (atual->destino) é igual à cidade de destino desejada (cidadeDestino). Se corresponder, isso significa que o nó atual representa a aresta que conecta a cidade de origem à cidade de destino. Se a cidade de destino for encontrada, a função retorna o peso (peso) da aresta correspondente. Esse peso é o valor armazenado em atual->peso, que representa a distância ou o custo para viajar da cidade de origem para a cidade de destino. Se a cidade de destino não for encontrada no nó atual, o ponteiro atual é atualizado para o próximo nó na lista e o loop continua. Se a função percorrer toda a lista de adjacência sem encontrar a cidade de destino, ela retorna -1. Isso indica que não há uma aresta direta conectando a cidade de origem à cidade de destino no grafo.

A função encontracaminhorec é uma função recursiva projetada para encontrar o caminho mais curto em um grafo ponderado, usando a técnica de backtracking. Ela recebe os seguintes parâmetros: posicao: Cidade atual no caminho. distancia: Distância acumulada até o momento. contador: Contador do número de cidades visitadas. visitado: Array para marcar as cidades já visitadas. caminhoAtual: Array que armazena o caminho atual. melhorCaminho: Array para armazenar o melhor caminho encontrado. numCidades: Número total de cidades no grafo. grafo: Ponteiro para o grafo ponderado. menorDistancia: Ponteiro para a menor distância encontrada. A função primeiro verifica se todas as cidades foram visitadas. Se sim, ela busca uma aresta que retorne à cidade de origem para formar um ciclo completo. Se um ciclo completo é encontrado e sua distância total é menor que a menorDistancia conhecida, o melhorCaminho e a menorDistancia são atualizados. A função então explora recursivamente as arestas saindo da posicao atual para encontrar todas as cidades ainda não visitadas.Para cada cidade não visitada e conectada, a função marca a cidade como visitada, adiciona-a ao caminhoAtual e chama encontraCaminhoRec para a próxima cidade. Após explorar uma cidade, a função desmarca a cidade para permitir a exploração de outros caminhos.

A função encontracaminho serve como uma função auxiliar para configurar e iniciar a busca recursiva pelo caminho mais curto. Recebe como parâmetro um ponteiro para o grafo ponderado e um array para armazenar o melhor caminho encontrado. Em suma ela realiza 5 etapas: 1- Aloca memória para o array visitado e caminhoAtual. 2-Inicializa a menorDistancia com um valor máximo. 3-Marca a cidade de origem como visitada e inicia o caminhoAtual a partir dela. 4-Chama encontraCaminhoRec para iniciar a busca pelo caminho mais curto. 5-Após a conclusão da busca, a função libera a memória alocada para os arrays visitado e caminhoAtual.

A função imprimeCaminho no código é responsável por imprimir o melhor caminho encontrado em um grafo ponderado, juntamente com a distância total desse caminho. A função começa imprimindo a cidade de origem do caminho. Na implementação padrão para problemas de caminho mais curto, geralmente se assume que o caminho começa e termina na mesma cidade (cidade 0), formando um ciclo. Em seguida, a função entra em um loop for para percorrer todas as cidades no array caminho. Cada cidade no caminho é impressa em sequência. Este loop não inclui a última cidade do caminho,

pois o caminho retorna à cidade de origem. Após imprimir todas as cidades no caminho, a função imprime novamente a cidade de origem para indicar o fechamento do ciclo. Por fim, a função imprime a distância total do caminho encontrado, que foi passada como parâmetro para a função.

A função ordenaLista no código é utilizada para ordenar as listas de adjacência de cada vértice em um grafo ponderado. Essa ordenação é feita com base no peso das arestas. A função começa com um loop for que percorre todas as cidades (ou vértices) do grafo. Para cada cidade, a função acessa sua lista de adjacência. Para cada lista de adjacência, a função primeiro verifica se a lista está vazia ou contém apenas um elemento. Em ambos os casos, a lista já está ordenada, e o loop continua para a próxima cidade. Se a lista contiver mais de um elemento, a função entra em um processo de ordenação: Inicializa uma nova lista ordenada (listaOrdenada), inicialmente vazia. Percorre os elementos da lista original, removendo cada elemento da lista original e inserindo-o na posição correta na lista ordenada. Para cada nó (atual) da lista original, a função determina a posição correta na lista ordenada com base no peso (peso) da aresta. Existem dois casos para a inserção: Inserção no Início: Se a lista ordenada estiver vazia ou se o peso do nó atual for menor ou igual ao peso do primeiro nó da lista ordenada, o nó é inserido no início da lista ordenada. Inserção no Meio ou Fim: Caso contrário, a função procura o local correto na lista ordenada (usando um loop while) e insere o nó atual nessa posição. Após ordenar todos os nós, a cabeça da lista original é atualizada para apontar para a cabeça da lista ordenada. Isso efetivamente substitui a lista original pela lista ordenada.

E por fim temos a função imprimeOrdenado que é responsável por imprimir as listas de adjacência de cada vértice em um grafo ponderado, após serem ordenadas pela função ordenaLista. Essa impressão oferece uma visualização clara da estrutura do grafo e de como os vértices estão conectados entre si. A função começa com um loop for que percorre todos os vértices (ou cidades) do grafo. Para cada vértice, a função imprime as adjacências relacionadas a esse vértice. Para cada vértice i, a função imprime uma mensagem inicial indicando que as adjacências do vértice i serão listadas. Isso é útil para identificar a qual vértice a lista de adjacência pertence. Em seguida, a função inicia um loop while que percorre a lista de adjacência do vértice i. Dentro deste loop, cada nó da lista (representando uma aresta) é processado. Para cada nó, a função imprime o destino da aresta (destino) e o peso da aresta (peso). Esta informação é essencial para entender como os vértices estão conectados e qual o custo associado à conexão. Após imprimir todos os nós da lista de adjacência de um vértice, a função imprime "NULL"ou uma indicação semelhante para sinalizar o final da lista de adjacência daquele vértice.

```c
#include "grafo.h" // Inclui o cabe alho "grafo.h" que cont m as
    declara  es de fun  es e estruturas.
#include <stdio.h> // Inclui a biblioteca padr o de entrada/sa da.
#include <stdlib.h> // Inclui a biblioteca padr o de aloca  o de mem ria.
#include <limits.h> // Inclui a biblioteca que fornece constantes de limites,
    incluindo INT_MAX.

// Fun  o para criar um novo n  (aresta) em um grafo ponderado.
No* criarNo(int destino, int peso) {
    // Aloca mem ria para um novo n .
    No* novoNo = (No*)malloc(sizeof(No));
    // Verifica se a aloca  o de mem ria foi bem-sucedida.
    if (novoNo == NULL) {
        exit(1); // Encerra o programa com c digo de erro 1 em caso de falha
            na aloca  o.
    }
    // Define o campo 'destino' do novo n  com o valor fornecido como destino
        .
    novoNo->destino = destino;
    // Define o campo 'peso' do novo n  com o valor fornecido como peso.
    novoNo->peso = peso;
    // Inicializa o campo 'proximo' do novo n  como nulo, indicando que
        inicialmente n o h  pr ximo n  na lista.
    novoNo->proximo = NULL;
    // Retorna o ponteiro para o novo n .
    return novoNo;
}
```

```
23
24  // Fun   o para criar uma nova lista de adjac ncia.
25  ListaAdj* criarListaAdj() {
26      // Aloca mem ria para uma nova lista de adjac ncia.
27      ListaAdj* lista = (ListaAdj*)malloc(sizeof(ListaAdj));
28      // Verifica se a aloca  o de mem ria foi bem-sucedida.
29      if (lista == NULL) {
30          exit(1); // Encerra o programa com c digo de erro 1 em caso de falha
                na aloca  o.
31      }
32      // Inicializa a cabe a da lista como nula.
33      lista->cabeca = NULL;
34      // Retorna o ponteiro para a nova lista de adjac ncia.
35      return lista;
36  }
37
38
39  // Fun   o para alocar dinamicamente mem ria para um novo grafo ponderado.
40  GrafoPonderado* alocarGrafo(int numCidades) {
41      // Aloca mem ria para uma nova estrutura GrafoPonderado.
42      GrafoPonderado* grafo = (GrafoPonderado*)malloc(sizeof(GrafoPonderado));
43      // Verifica se a aloca  o de mem ria foi bem-sucedida.
44      if (grafo == NULL) {
45          exit(1); // Encerra o programa com c digo de erro 1 em caso de falha
                na aloca  o.
46      }
47      // Inicializa o n mero de cidades na estrutura do grafo.
48      grafo->numCidades = numCidades;
49      // Aloca mem ria para a lista de adjac ncia do grafo.
50      grafo->listaAdjacencia = (ListaAdj*)malloc(numCidades * sizeof(ListaAdj));
51      // Verifica se a aloca  o da lista de adjac ncia foi bem-sucedida.
52      if (grafo->listaAdjacencia == NULL) {
53          // Tratamento de erro: libera a mem ria alocada para a estrutura do
                grafo e encerra o programa com c digo de erro 1.
54          free(grafo);
55          exit(1);
56      }
57      // Inicializa a lista de adjac ncia, definindo a cabe a de cada lista
            como nula.
58      for (int i = 0; i < numCidades; i++) {
59          grafo->listaAdjacencia[i].cabeca = NULL;
60      }
61      return grafo;
62  }
63
64  // Desaloca Grafo
65  void desalocarGrafo(GrafoPonderado* grafo) {
66      // Verifica se o ponteiro para o grafo  n o    nulo.
67      if (grafo != NULL) {
68          // Loop externo para percorrer todas as cidades do grafo.
69          for (int i = 0; i < grafo->numCidades; i++) {
70              // Inicializa um ponteiro para o primeiro  n   na lista de
                    adjac ncia da cidade atual.
71              No* atual = grafo->listaAdjacencia[i].cabeca;
72              // Loop interno para liberar a mem ria de todos os  n  s na lista
                    de adjac ncia da cidade atual.
73              while (atual != NULL) {
74                  // Armazena o pr ximo  n   antes de liberar a mem ria do  n
                        atual.
75                  No* proximo = atual->proximo;
76                  // Libera a mem ria alocada para o  n   atual.
77                  free(atual);
```

```
78                  // Atualiza o ponteiro para o pr ximo  n   na lista.
79                  atual = proximo;
80              }
81          }
82          // Libera a mem ria alocada para a lista de adjac ncia.
83          free(grafo->listaAdjacencia);
84          // Libera a mem ria alocada para a estrutura GrafoPonderado.
85          free(grafo);
86      }
87  }
88
89  // Ler grafo
90  void leGrafo(GrafoPonderado* grafo) {
91      // Obt m o n mero de cidades do grafo.
92      int numCidades = grafo->numCidades;
93      // Loop externo para percorrer todas as cidades do grafo.
94      for (int i = 0; i < numCidades; i++) {
95          // Loop interno para ler as dist ncias entre a cidade 'i' e todas as
                outras cidades.
96          for (int j = 0; j < numCidades; j++) {
97              // Declara  o das vari veis para origem, destino e dist ncia
                    entre as cidades.
98              int origem, destino, distancia;
99              // L   os valores de origem, destino e dist ncia da aresta entre
                    as cidades.
100             scanf("%d %d %d", &origem, &destino, &distancia);
101             // Atualiza a lista de adjac ncia com a dist ncia lida.
102             No* novoNo = criarNo(destino, distancia);
103             novoNo->proximo = grafo->listaAdjacencia[origem].cabeca;
104             grafo->listaAdjacencia[origem].cabeca = novoNo;
105         }
106     }
107 }
108
109 // Fun   o para obter a dist ncia entre duas cidades no grafo.
110 int obterDistancia(GrafoPonderado* grafo, int cidadeOrigem, int cidadeDestino)
        {
111     // Inicializa um ponteiro do tipo No, apontando para a cabe a da lista de
            adjac ncia da cidade de origem no grafo.
112     No* atual = grafo->listaAdjacencia[cidadeOrigem].cabeca;
113     // Inicia um loop while que continua at   que o ponteiro 'atual' seja NULL
        , indicando o fim da lista.
114     while (atual != NULL) {
115         // Verifica se o n   atual na lista de adjac ncia    a cidade de
                destino.
116         if (atual->destino == cidadeDestino) {
117             // Se for, retorna o peso (ou dist ncia) associado a esse n  ,
                    que    a dist ncia at   a cidade de destino.
118             return atual->peso;
119         }
120         // Atualiza o ponteiro 'atual' para o pr ximo  n   na lista de
                adjac ncia.
121         atual = atual->proximo;
122     }
123     // Se a cidade de destino  n  o for encontrada na lista de adjac ncia da
            cidade de origem, retorna -1.
124     return -1;
125 }
126
127 // Fun   o recursiva para encontrar o melhor caminho em um grafo ponderado
        usando lista de adjac ncia.
128 void encontraCaminhoRec(int posicao, int distancia, int contador, int*
```

```
    visitado, int* caminhoAtual, int* melhorCaminho, int numCidades,
    GrafoPonderado* grafo, int* menorDistancia) {
129     // Verifica se todas as cidades foram visitadas (exceto a cidade de origem
        ).
130     if (contador == numCidades - 1) {
131         // Procura por uma aresta que retorne    cidade de origem (posi   o
            0) para formar um ciclo completo.
132         No* vizinho = grafo->listaAdjacencia[posicao].cabeca;
133         while (vizinho != NULL) {
134             int cidadeDestino = vizinho->destino;
135             // Se encontrarmos um ciclo completo com dist ncia total menor
                que a menor dist ncia conhecida, atualizamos o melhor caminho
                 e a menor dist ncia.
136             if (cidadeDestino == 0 && vizinho->peso != 0 && distancia +
                vizinho->peso < *menorDistancia) {
137                 *menorDistancia = distancia + vizinho->peso;
138                 for (int i = 0; i < numCidades; i++) {
139                     melhorCaminho[i] = caminhoAtual[i];
140                 }
141             }
142             vizinho = vizinho->proximo;
143         }
144         // Retorna da recurs o.
145         return;
146     }
147     // Explora as arestas saindo da posi   o atual para encontrar todas as
        cidades ainda n  o visitadas.
148     for (No* vizinho = grafo->listaAdjacencia[posicao].cabeca; vizinho != NULL
        ; vizinho = vizinho->proximo) {
149         int proximaCidade = vizinho->destino;
150         // Verifica se a pr xima cidade n o foi visitada e se a aresta atual
                n o    uma aresta sem peso (peso != 0).
151         if (visitado[proximaCidade] == 0 && vizinho->peso != 0) {
152             // Marca a pr xima cidade como visitada e adiciona ao caminho
                atual.
153             visitado[proximaCidade] = 1;
154             caminhoAtual[contador] = proximaCidade;
155             // Chama a fun   o recursivamente para a pr xima cidade.
156             encontraCaminhoRec(proximaCidade, distancia + vizinho->peso,
                contador + 1, visitado, caminhoAtual, melhorCaminho,
                numCidades, grafo, menorDistancia);
157             // Desmarca a cidade ap s a chamada recursiva para permitir a
                explora   o de outros caminhos.
158             visitado[proximaCidade] = 0;
159         }
160     }
161 }
162
163 // Fun   o para encontrar o caminho mais curto em um grafo ponderado usando
    for a bruta.
164 void encontraCaminho(GrafoPonderado* grafo, int* melhorCaminho) {
165     int numCidades = grafo->numCidades;
166     // Aloca um array para controlar as cidades visitadas.
167     int* visitado = (int*)malloc(numCidades * sizeof(int));
168     // Aloca um array para rastrear o caminho atual.
169     int* caminhoAtual = (int*)malloc(numCidades * sizeof(int));
170     // Inicializa a menorDistancia com um valor m ximo.
171     int menorDistancia = INT_MAX;
172     // Marca a cidade de origem (0) como visitada e inicia o caminho atual.
173     for (int i = 0; i < numCidades; i++) {
174         visitado[i] = 0;
175     }
```

```
176     visitado[0] = 1;
177     caminhoAtual[0] = 0;
178     // Chama a fun   o encontraCaminhoRec para encontrar o caminho mais curto
            .
179     encontraCaminhoRec(0, 0, 0, visitado, caminhoAtual, melhorCaminho,
            numCidades, grafo, &menorDistancia);
180     // Libera a mem ria alocada para os arrays de controle.
181     free(visitado);
182     free(caminhoAtual);
183 }
184
185 // Fun   o para imprimir o caminho mais curto encontrado e a dist ncia total
        .
186 void imprimeCaminho(GrafoPonderado* grafo, int* caminho, int distancia) {
187     // Imprime a cidade de origem (0) para iniciar o caminho.
188     printf("Melhor caminho: 0 ");
189     // Percorre as cidades do caminho, exceto a  ltima , e as imprime.
190     for (int i = 0; i < grafo->numCidades - 1; i++) {
191         printf("%d ", caminho[i]);
192     }
193     // Imprime a cidade de origem (0) novamente para completar o ciclo.
194     printf("0\n");
195     // Imprime a dist ncia total do caminho encontrado.
196     printf("Melhor distancia: %d\n", distancia);
197 }
198
199 // Fun   o para ordenar as listas de adjac ncia de um grafo ponderado.
200 void ordenaLista(GrafoPonderado* grafo) {
201     // Loop externo para percorrer todas as cidades do grafo.
202     for (int i = 0; i < grafo->numCidades; i++) {
203         // Obt m o ponteiro para a lista de adjac ncia da cidade atual.
204         ListaAdj* lista = &grafo->listaAdjacencia[i];
205         // Verifica se a lista est  vazia ou tem apenas um elemento, o que
                significa que j  est  ordenada.
206         if (lista->cabeca == NULL || lista->cabeca->proximo == NULL) {
207             continue; // Pula para a pr xima itera   o do loop se a lista
                    j  estiver ordenada.
208         }
209         // Inicializa uma lista tempor ria para armazenar os elementos
                ordenados.
210         ListaAdj listaOrdenada;
211         listaOrdenada.cabeca = NULL;
212         // Percorre os elementos da lista original.
213         No* atual = lista->cabeca;
214         while (atual != NULL) {
215             // Armazena o pr ximo n  antes de remov -lo da lista original.
216             No* proximo = atual->proximo;
217             // Insere o n  na lista ordenada.
218             if (listaOrdenada.cabeca == NULL || atual->peso <= listaOrdenada.
                    cabeca->peso) {
219                 // Caso especial: insere no in cio da lista ordenada.
220                 atual->proximo = listaOrdenada.cabeca;
221                 listaOrdenada.cabeca = atual;
222             } else {
223                 // Procura o local correto na lista ordenada para inserir o
                        n .
224                 No* anterior = NULL;
225                 No* atualOrdenada = listaOrdenada.cabeca;
226                 while (atualOrdenada != NULL && atual->peso > atualOrdenada->
                        peso) {
227                     anterior = atualOrdenada;
228                     atualOrdenada = atualOrdenada->proximo;
```

```
                }
                // Insere o n  na posi  o correta.
                anterior->proximo = atual;
                atual->proximo = atualOrdenada;
            }
            // Move para o pr ximo n  na lista original.
            atual = proximo;
        }
        // Atualiza a cabe a da lista original para apontar para a lista
            ordenada.
        lista->cabeca = listaOrdenada.cabeca;
    }
}
// Fun  o para imprimir as listas de adjac ncia ordenada.
void imprimeOrdenado(GrafoPonderado* grafo) {
    // Loop para percorrer todos os v rtices do grafo.
    for (int i = 0; i < grafo->numCidades; i++) {
        // Imprime mensagem indicando a lista de adjac ncias do v rtice i.
        printf("Adjacencias do vertice %d: ", i);
        // Inicializa um ponteiro para o primeiro n  na lista de adjac ncias
            do v rtice i.
        No* atual = grafo->listaAdjacencia[i].cabeca;
        // Loop para percorrer todos os n s na lista de adjac ncias do
            v rtice i.
        while (atual != NULL) {
            // Imprime as informa  es do n  atual, incluindo o destino e o
                peso da aresta.
            printf("(%d, %d) -> ", atual->destino, atual->peso);
            // Atualiza o ponteiro para apontar para o pr ximo n  na lista
                de adjac ncias.
            atual = atual->proximo;
        }
        // Imprime "NULL" para indicar o final da lista de adjac ncias do
            v rtice i.
        printf("NULL\n");
    }
}
```

Código 2: grafo.c

2.3 Execução

O arquivo main.c é o ponto de entrada do programa que lida com grafos ponderados. A função main é o ponto de partida do programa. Ela executa as seguintes operações: Leitura do Número de Cidades: scanf(”Alocação do Grafo: GrafoPonderado* grafo = alocarGrafo(numCidades);: Aloca um grafo ponderado com um número especificado de cidades. Leitura das Informações do Grafo: leGrafo(grafo);: Lê as informações das distâncias entre as cidades e as armazena no grafo. Ordenação das Listas de Adjacências: ordenaLista(grafo);: Ordena as listas de adjacências do grafo. Alocação do Melhor Caminho: int* melhorCaminho = (int*)malloc((numCidades - 1) * sizeof(int));: Aloca memória para armazenar o melhor caminho encontrado. Encontrando o Caminho Mais Curto: encontraCaminho(grafo, melhorCaminho);: Encontra o caminho mais curto através do grafo. Cálculo da Distância Total: Um loop for é usado para calcular a distância total percorrida ao somar as distâncias entre as cidades do caminho. Adicionar Distância do Último para o Primeiro Vértice: distancia += obterDistancia(grafo, melhorCaminho[numCidades - 1], melhorCaminho[0]);: Adiciona a distância entre a última cidade e a cidade de origem para fechar o ciclo. Imprimir Lista de Adjacências e Caminho: imprimeOrdenado(grafo);: Imprime a lista de adjacências ordenada. imprimeCaminho(grafo, melhorCaminho, distancia);: Imprime o caminho mais curto encontrado e a distância total. Liberação da Memória: free(melhorCaminho);: Libera a memória alocada para o vetor do melhor caminho. desalocarGrafo(grafo);: Libera a memória alocada para o grafo. Retorno da Função main: return 0;: Retorna 0 para indicar que o programa foi executado com sucesso.

O main.c gerencia o fluxo geral do programa, tratando da leitura de dados, processamento do grafo (como alocação, leitura, ordenação, busca do melhor caminho) e, finalmente, da impressão dos resultados e liberação de recursos. É a parte do programa que orquestra as operações sobre o grafo, utilizando as funções definidas em grafo.c e as estruturas definidas em grafo.h.

```c
#include "grafo.h" // Inclui a defini  o das estruturas e fun  es do grafo.
#include <stdio.h> // Inclui a biblioteca padr o para entrada/sa da.
#include <stdlib.h> // Inclui a biblioteca padr o de aloca  o de mem ria.
int main() {
    int numCidades;
    scanf("%d", &numCidades); // L  o n mero de cidades a partir da entrada padr o.
    // Aloca espa o na mem ria para um grafo ponderado com 'numCidades'.
    GrafoPonderado* grafo = alocarGrafo(numCidades);
    // L  as informa  es das dist ncias entre as cidades e as armazena no grafo.
    leGrafo(grafo);
    // Ordena as listas de adjac ncias
    ordenaLista(grafo);
    // Aloca mem ria para armazenar o melhor caminho encontrado, exceto a cidade de origem.
    int* melhorCaminho = (int*)malloc((numCidades - 1) * sizeof(int));
    // Encontra o caminho mais curto atrav s do grafo.
    encontraCaminho(grafo, melhorCaminho);
    int distancia = 0; // Inicializa a vari vel que armazenar  a dist ncia total do caminho.
    // Calcula a dist ncia total percorrida ao somar as dist ncias entre as cidades do caminho.
    for (int i = 0; i < numCidades - 1; i++) {
        int cidadeOrigem = melhorCaminho[i];
        int cidadeDestino = melhorCaminho[i + 1];
        distancia += obterDistancia(grafo, cidadeOrigem, cidadeDestino);
    }
    // Adiciona a dist ncia entre a  ltima  cidade e a cidade de origem para fechar o ciclo.
    distancia += obterDistancia(grafo, melhorCaminho[numCidades - 1], melhorCaminho[0]);
    //Imprime a lista de adjacencias
    imprimeOrdenado(grafo);
    // Imprime o caminho mais curto encontrado e a dist ncia total.
    imprimeCaminho(grafo, melhorCaminho, distancia);
    // Libera a mem ria alocada para o vetor do melhor caminho.
    free(melhorCaminho);
    // Libera a mem ria alocada para o grafo.
    desalocarGrafo(grafo);
    return 0; // Retorna 0 para indicar que o programa foi executado com sucesso.
}
```

Código 3: main.c

3 Testes

Os testes foram realizados com os arquivos disponibilizados pelo professor da disciplina. No caso foram disponibilizados 5 arquivos com as entradas e 5 arquivos com as saídas esperadas. Para o primeiro caso, as entradas sao

```
4
0 0 0
0 1 10
```

```
0 2 15
0 3 20
1 0 10
1 1 0
1 2 35
1 3 25
2 0 15
2 1 35
2 2 0
2 3 30
3 0 20
3 1 25
3 2 30
3 3 0
```

e a saída esperada deve ser

```
Adjacencias do vertice 0: (0, 0) -> (1, 10) -> (2, 15) -> (3, 20) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 10) -> (3, 25) -> (2, 35) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 15) -> (3, 30) -> (1, 35) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 20) -> (1, 25) -> (2, 30) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 80
```

Para o segundo caso, as entradas sao˜

```
6
0 0 0
0 1 1
0 2 2
0 3 1
0 4 1
0 5 2
1 0 1
1 1 0
1 2 7
1 3 1
1 4 4
1 5 3
2 0 2
2 1 7
2 2 0
2 3 3
2 4 1
2 5 1
3 0 1
3 1 1
3 2 3
3 3 0
3 4 8
3 5 1
4 0 1
4 1 2
4 2 1
4 3 8
4 4 0
4 5 1
5 0 2
```

```
5 1 3
5 2 1
5 3 1
5 4 1
5 5 0
```

e a saída esperada deve ser

```
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (3, 1) -> (4, 1) -> (2, 2) -> (5, 2) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (3, 1) -> (5, 3) -> (4, 4) -> (2, 7) -> NULL
Adjacencias do vertice 2: (2, 0) -> (4, 1) -> (5, 1) -> (0, 2) -> (3, 3) -> (1, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 1) -> (1, 1) -> (5, 1) -> (2, 3) -> (4, 8) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (2, 1) -> (5, 1) -> (1, 2) -> (3, 8) -> NULL
Adjacencias do vertice 5: (5, 0) -> (2, 1) -> (3, 1) -> (4, 1) -> (0, 2) -> (1, 3) -> NULL
Melhor caminho: 0 1 3 5 2 4 0
Melhor distancia: 6
```

Para o terceiro caso, as entradas são

```
4
0 0 0
0 1 1
0 2 1
0 3 3
1 0 1
1 1 0
1 2 4
1 3 5
2 0 1
2 1 4
2 2 0
2 3 6
3 0 3
3 1 5
3 2 6
3 3 0
```

e a saída esperada deve ser

```
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (2, 1) -> (3, 3) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (2, 4) -> (3, 5) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 1) -> (1, 4) -> (3, 6) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 5) -> (2, 6) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 13
```

Para o quarto caso, as entradas são

```
5
0 0 0
0 1 2
0 2 0
0 3 3
0 4 6
1 0 2
1 1 0
1 2 4
1 3 3
1 4 0
2 0 0
```

```
2 1 4
2 2 0
2 3 7
2 4 3
3 0 3
3 1 3
3 2 7
3 3 0
3 4 3
4 0 6
4 1 0
4 2 3
4 3 3
4 4 0
```

e a saída esperada deve ser

```
Adjacencias do vertice 0: (0, 0) -> (2, 0) -> (1, 2) -> (3, 3) -> (4, 6) -> NULL
Adjacencias do vertice 1: (1, 0) -> (4, 0) -> (0, 2) -> (3, 3) -> (2, 4) -> NULL
Adjacencias do vertice 2: (0, 0) -> (2, 0) -> (4, 3) -> (1, 4) -> (3, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 3) -> (4, 3) -> (2, 7) -> NULL
Adjacencias do vertice 4: (1, 0) -> (4, 0) -> (2, 3) -> (3, 3) -> (0, 6) -> NULL
Melhor caminho: 0 1 2 4 3 0
Melhor distancia: 15
```

Para o quinto caso, as entradas sao

```
5
0 0 0
0 1 5
0 2 10
0 3 0
0 4 1
1 0 5
1 1 0
1 2 0
1 3 10
1 4 1
2 0 10
2 1 0
2 2 0
2 3 2
2 4 1
3 0 0
3 1 10
3 2 2
3 3 0
3 4 1
4 0 1
4 1 1
4 2 1
4 3 1
4 4 0
```

e a saída esperada deve ser

```
Adjacencias do vertice 0: (0, 0) -> (3, 0) -> (4, 1) -> (1, 5) -> (2, 10) -> NULL
Adjacencias do vertice 1: (1, 0) -> (2, 0) -> (4, 1) -> (0, 5) -> (3, 10) -> NULL
Adjacencias do vertice 2: (1, 0) -> (2, 0) -> (4, 1) -> (3, 2) -> (0, 10) -> NULL
```

```
Adjacencias do vertice 3: (0, 0) -> (3, 0) -> (4, 1) -> (2, 2) -> (1, 10) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (1, 1) -> (2, 1) -> (3, 1) -> NULL
Melhor caminho: 0 1 3 2 4 0
Melhor distancia: 19
```

O programa atendeu a todos os testes, menos ao teste 5 que retorna o melhor caminho como 0 4 2 3 1 0, sendo que deveria ser 0 1 3 2 4 0. Todavia, após conversa com o professor da disciplina foi decidido que poderia ficar assim mesmo, que não haveria perca de pontos pois todo o resto está certo.

4 Análise

Nesse programa, assim como no trabalho anterior, foi implementada uma abordagem de força bruta utilizando recursividade para encontrar o caminho mais curto, o que significa que ele explora todas as possíveis combinações de cidades para encontrar a solução. Esse trabalho trouxe um nível de dificuldade um pouco maior por ter que utilizar listas encadeadas, mas possuí a mesma questão do primeiro trabalho por usar recursividade. Para conjuntos de cidades pequenos, a abordagem utilizando recursividade pode ser aceitável e fornecer soluções precisas. No entanto, para um grande número de cidades, a execução do programa pode levar um tempo impraticável, tornando-o ineficiente. Para resolver o problema de forma eficiente em conjuntos de cidades maiores, seriam necessárias abordagens mais sofisticadas, como algoritmos baseados em heurísticas. Esse trabalho demonstra a complexidade dos testes quando levados a um número maior de entradas. Com isso, prova-se a importância da análise de complexidades.

5 Considerações Finais

Desenvolver esse trabalho foi bem significante para mim, pois aprimorou meu conhecimento sobre funções recursivas, ordenação e listas encadeadas. Além da dificuldade do trabalho anterior de ter que desenvolver o trabalho sozinho tive a dificuldade de ter que fazer o trabalho pós operado, o que foi bem desgastante para mim. Fora isso, gostei bastante de fazer o trabalho e conclui-lo é bem gratificante para mim.

Referências

CELES,Waldemar; CERQUEIRA,Renato; RANGEL,Jose Lucas. Introdução a Estruturas de Dados: com técnicas de programação em C.. Rio de Janeiro: Elsevier 2004. 293 p.