

# Lista de exercícios Teoria dos Grafos

Nome: Pedro Henrique dos Santos Oliveira - RGM: 31274820

Curso: Ciência da Computação - Turma: 4A

[pedrohenrique\\_santos@live.com](mailto:pedrohenrique_santos@live.com)

Universidade da Cidade de São Paulo (UNICID) - Rua Cesário Galeno, 448/475

São Paulo – SP – Brasil – CEP: 03071-000

***Abstract.** In this project, a list of exercises was developed for practicing Graph Theory in C language, using DFS (Deep Search) techniques. The problem involves manipulating graphs, with user input containing the graph representation and operations on it, such as inserting vertices, searching for source and destination vertices, and removing vertices. The aim is to apply the theoretical concepts of Graph Theory to solve practical problems, demonstrating the ability to divide a problem into logical parts to achieve an effective solution. This project demonstrates the application of knowledge acquired in Professor Juliano Ratusznei's classes and the ability to divide a problem into logical parts to achieve an effective solution.*

**Resumo.** Neste projeto, foi desenvolvida uma lista de exercícios para prática de Teoria dos Grafos em linguagem C, utilizando técnicas de DFS (Busca em Profundidade). O problema envolve a manipulação de grafos, com entrada do usuário contendo a representação do grafo e operações sobre ele, como inserção de vértices, busca por vértices de origem e destino, e remoção de vértices. O objetivo é aplicar os conceitos teóricos de Teoria dos Grafos na resolução de problemas práticos, demonstrando a capacidade de dividir um problema em partes lógicas para alcançar uma solução eficaz. Este projeto demonstra a aplicação de conhecimentos adquiridos nas aulas do professor Juliano Ratusznei e a capacidade de dividir um problema em partes lógicas para atingir uma solução eficaz.

## Descritiva do problema a ser solucionado

O usuário entra com:

a) Representação do grafo:

- Quantidade de vértices
- Ligações entre os vértices

b) Insere os vértices de origem e destino. Portanto pode ser escolhido qualquer um dos vértices pertencentes ao grafo.

## Codificação:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define MAX_VERTICES 100

typedef struct {
    int **adj_matrix;
    int num_vertices;
} Grafo;

int num_vertices; // Variável global para o número de vértices

// Função para criar um grafo com um número específico de vértices
Grafo *criarGrafo(int num_vertices) {
    Grafo *grafo = (Grafo *)malloc(sizeof(Grafo));
    grafo->num_vertices = num_vertices;

    // Aloca a matriz de adjacência e inicializa com zeros
    grafo->adj_matrix = (int **)malloc(num_vertices * sizeof(int *));
    int i;
    for (i = 0; i < num_vertices; i++) {
        grafo->adj_matrix[i] = (int *)calloc(num_vertices, sizeof(int));
    }

    return grafo;
}

// Função para adicionar uma aresta entre dois vértices
void adicionarAresta(Grafo *grafo, int vertice1, int vertice2) {
    grafo->adj_matrix[vertice1][vertice2] = 1;
    grafo->adj_matrix[vertice2][vertice1] = 1;
}

// Função para liberar a memória alocada para o grafo
void liberarGrafo(Grafo *grafo) {
    int i;
    for (i = 0; i < grafo->num_vertices; i++) {
        free(grafo->adj_matrix[i]);
    }
    free(grafo->adj_matrix);
    free(grafo);
}

// Função para imprimir a matriz de adjacência do grafo
void imprimirGrafo(Grafo *grafo) {
    printf("Matriz de Adjacência:\n");
    int i, j;
    for (i = 0; i < grafo->num_vertices; i++) {
        for (j = 0; j < grafo->num_vertices; j++) {
```

```

        printf("%d ", grafo->adj_matrix[i][j]);
    }
    printf("\n");
}

// Função para verificar se há um caminho entre origem e destino usando DFS
int temCaminho(Grafo *grafo, int origem, int destino, int *visitados, int
*caminho, int passo) {
    caminho[passo] = origem; // Adiciona o vértice ao caminho
    if (origem == destino) {
        return 1; // Há um caminho entre origem e destino
    }

    visitados[origem] = 1; // Marcar o vértice como visitado

    int i;
    for (i = 0; i < grafo->num_vertices; i++) {
        if (grafo->adj_matrix[origem][i] && !visitados[i]) {
            if (temCaminho(grafo, i, destino, visitados, caminho, passo +
1)) {
                return 1;
            }
        }
    }

    // Se não houver caminho, desmarcar o vértice
    visitados[origem] = 0;
    return 0;
}

int main() {
    setlocale(LC_ALL, "Portuguese");

    int opcao, origem, destino;
    Grafo *grafo = NULL;
    int i, j; // Variáveis de controle

    do {
        // Menu de opções
        printf("\n[1] Novo Grafo\n[2] Visualizar Grafo\n[3] Visualizar
Caminho\n[0] Finalizar\n");
        printf("Escolha uma opção: ");
        scanf("%d", &opcao);

        switch (opcao) {
            case 1:
                // Criar um novo grafo
                if (grafo != NULL) {
                    liberarGrafo(grafo); // Libera o grafo anterior se
existir

                }
                printf("\nInsira o número de vértices: \n");
                scanf("%d", &num_vertices); // Captura o número de vértices
                grafo = criarGrafo(num_vertices); // Cria o grafo com o
número especificado
                printf("\nInsira as conexões entre os vértices (1 para
conectado, 0 para não conectado):\n");
                for (i = 0; i < num_vertices; i++) {
                    for (j = i; j < num_vertices; j++) {

```

```

        if (i == j) {
            printf("%d para %d: ", i + 1, j + 1);
            scanf("%d", &grafo->adj_matrix[i][j]);
        } else {
            printf("%d para %d: ", i + 1, j + 1);
            scanf("%d", &grafo->adj_matrix[i][j]);
            grafo->adj_matrix[j][i] = grafo->adj_matrix[i][j]; // A matriz é simétrica
        }
    }
    break;
case 2:
    // Visualizar o grafo
    if (grafo != NULL) {
        system("cls || clear"); // Limpa a tela
        imprimirGrafo(grafo);
    } else {
        system("cls || clear");
        printf("!!!Grafo não foi criado ainda!!!\n");
    }
    break;
case 3:
    // Visualizar caminho entre vértices
    if (grafo != NULL) {
        system("cls || clear");
        printf("\nInsira o vértice de origem: ");
        scanf("%d", &origem);
        printf("Insira o vértice de destino: ");
        scanf("%d", &destino);

        // Verificar se há um caminho entre origem e destino
        int *visitados = (int *)calloc(grafo->num_vertices, sizeof(int));
        int *caminho = (int *)calloc(grafo->num_vertices, sizeof(int));

        if (temCaminho(grafo, origem - 1, destino - 1, visitados, caminho, 0)) {
            system("cls || clear");
            printf("Existe um caminho entre o vértice %d e o vértice %d.\n", origem, destino);
            printf("\nCaminho: \n");
            printf("%d", origem);
            int i;
            for (i = 1; caminho[i] != destino - 1; i++) {
                printf(" -> %d", caminho[i] + 1);
            }
            printf(" -> %d\n", destino);
        } else {
            // Se não houver caminho ou os vértices estiverem fora do intervalo válido
            if (origem < 1 || origem > grafo->num_vertices || destino < 1 || destino > grafo->num_vertices) {
                printf("\nNão há caminho entre o vértice %d e o vértice %d.\n", origem, destino);
            } else {
                printf("\nNão há caminho entre o vértice %d e o vértice %d.\n", origem, destino);
            }
        }
    }
}

```

```

        free(visitados);
        free(caminho);
    } else {
        system("cls || clear");
        printf("\n!!!!Grafo não foi criado ainda!!!\n");
    }
    break;
case 0:
    // Finalizar o programa
    if (grafo != NULL) {
        liberarGrafo(grafo); // Libera a memória alocada para o
grafo
    }
    system("cls || clear");
    printf("\nPrograma finalizado.\n");
    break;
default:
    system("cls || clear");
    printf("\nOpção inválida. Tente novamente.\n");
}
} while (opcao != 0);

return 0;
}

```

## 1. Estruturas de Dados e Constantes

O código começa com a inclusão de bibliotecas padrão em C, como `<stdio.h>` e `<stdlib.h>`, para manipulação de entrada/saída e alocação de memória, respectivamente. Além disso, inclui `<locale.h>` para definir a localidade do programa.

Em seguida, define-se uma constante `MAX_VERTICES` com valor 100, representando o número máximo de vértices permitidos no grafo.

A estrutura `Grafo` é definida usando a palavra-chave `typedef`, consistindo em uma matriz de adjacência (`adj_matrix`) e o número de vértices (`num_vertices`). Esta estrutura é fundamental para representar um grafo no programa.

A variável global `num_vertices` é declarada para armazenar o número de vértices em uso, enquanto as funções e operações subsequentes operam sobre esta variável.

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define MAX_VERTICES 100

typedef struct {
    int **adj_matrix;
    int num_vertices;
} Grafo;

int num_vertices; // Variável global para o número de vértices

```

## 2. Funções Principais

### 2.1. Função criarGrafo

Esta função aloca dinamicamente memória para um novo grafo com um número específico de vértices. Ela retorna um ponteiro para a estrutura Grafo.

criarGrafo(int num\_vertices):

- Permite a criação de um novo grafo com um número específico de vértices (Fig. 02).
- Aloca dinamicamente memória para a estrutura Grafo.
- Inicializa o número de vértices do grafo com o valor fornecido.
- Aloca memória para a matriz de adjacência do grafo, inicializando-a com zeros.
- Retorna um ponteiro para o grafo recém-criado.

```
// Função para criar um grafo com um número específico de vértices
Grafo *criarGrafo(int num_vertices) {
    Grafo *grafo = (Grafo *)malloc(sizeof(Grafo));
    grafo->num_vertices = num_vertices;

    // Aloca a matriz de adjacência e inicializa com zeros
    grafo->adj_matrix = (int **)malloc(num_vertices * sizeof(int *));
    int i;
    for (i = 0; i < num_vertices; i++) {
        grafo->adj_matrix[i] = (int *)calloc(num_vertices, sizeof(int));
    }

    return grafo;
}
```

### 2.2. Função adicionarAresta

Esta função adiciona uma aresta entre dois vértices em um grafo, atualizando a matriz de adjacência. Ela recebe como parâmetros o grafo, os vértices de origem e destino, e define a conexão entre eles na matriz de adjacência.

adicionarAresta(Grafo grafo, int vertice1, int vertice2):

- Adiciona uma aresta entre dois vértices no grafo.
- Atualiza a matriz de adjacência para refletir a conexão entre os vértices.
- A conexão entre vertice1 e vertice2 é estabelecida definindo os valores correspondentes na matriz como 1.

```
// Função para adicionar uma aresta entre dois vértices
void adicionarAresta(Grafo *grafo, int vertice1, int vertice2) {
    grafo->adj_matrix[vertice1][vertice2] = 1;
    grafo->adj_matrix[vertice2][vertice1] = 1;
}
```

### 2.3. Função liberarGrafo

Esta função libera a memória alocada dinamicamente para um grafo, desalocando a matriz de adjacência e a própria estrutura Grafo.

liberarGrafo(Grafo grafo):

- Libera a memória alocada para o grafo e sua matriz de adjacência.
- Desaloca cada linha da matriz de adjacência.
- Desaloca o vetor de ponteiros da matriz.
- Desaloca a estrutura Grafo em si.

```
// Função para liberar a memória alocada para o grafo
void liberarGrafo(Grafo *grafo) {
    int i;
    for (i = 0; i < grafo->num_vertices; i++) {
        free(grafo->adj_matrix[i]);
    }
    free(grafo->adj_matrix);
    free(grafo);
}
```

### 2.4. Função imprimirGrafo

Esta função imprime a matriz de adjacência do grafo na saída padrão, facilitando a visualização da estrutura do grafo (Fig. 3).

imprimirGrafo(Grafo grafo):

- Imprime a matriz de adjacência do grafo na saída padrão.
- Itera sobre cada linha e coluna da matriz, imprimindo os valores correspondentes.
- Facilita a visualização da estrutura do grafo para o usuário.

```
// Função para imprimir a matriz de adjacência do grafo
void imprimirGrafo(Grafo *grafo) {
    printf("Matriz de Adjacência:\n");
    int i, j;
    for (i = 0; i < grafo->num_vertices; i++) {
        for (j = 0; j < grafo->num_vertices; j++) {
            printf("%d ", grafo->adj_matrix[i][j]);
        }
        printf("\n");
    }
}
```

## 2.5. Função temCaminho

Esta função verifica se há um caminho entre dois vértices em um grafo usando a busca em profundidade (DFS). Ela retorna 1 se um caminho é encontrado e 0 caso contrário. Utiliza-se um vetor de visitados para evitar ciclos durante a travessia do grafo.

temCaminho(Grafo \*grafo, int origem, int destino, int visitados, int caminho, int passo):

- Verifica se há um caminho entre dois vértices no grafo usando busca em profundidade (DFS).
- Armazena o caminho percorrido na matriz caminho.
- Marca os vértices visitados no vetor visitados.
- Retorna 1 se um caminho é encontrado entre os vértices de origem e destino.
- Retorna 0 caso contrário.

```
// Função para verificar se há um caminho entre origem e destino usando DFS
int temCaminho(Grafo *grafo, int origem, int destino, int *visitados, int *caminho, int passo)
{
    caminho[passo] = origem; // Adiciona o vértice ao caminho
    if (origem == destino) {
        return 1; // Há um caminho entre origem e destino
    }

    visitados[origem] = 1; // Marcar o vértice como visitado

    int i;
    for (i = 0; i < grafo->num_vertices; i++) {
        if (grafo->adj_matrix[origem][i] && !visitados[i]) {
            if (temCaminho(grafo, i, destino, visitados, caminho, passo + 1)) {
                return 1;
            }
        }
    }

    // Se não houver caminho, desmarcar o vértice
    visitados[origem] = 0;
    return 0;
}
```

## 2.6. Função main

A função principal main controla a interação com o usuário e o menu de opções do programa. Ela permite criar um novo grafo, visualizar o grafo, verificar a existência de um caminho entre vértices e finalizar o programa. As opções selecionadas pelo usuário são tratadas por meio de um loop do-while e um bloco switch-case.

int main():

- Função principal do programa.
- Inicializa as variáveis opcao, origem, destino e grafo.
- Define duas variáveis de controle i e j.



- Configura o ambiente para utilizar a localização em português.
- Utiliza um loop do-while para apresentar continuamente o menu de opções ao usuário e processar suas escolhas até que ele decida finalizar o programa.

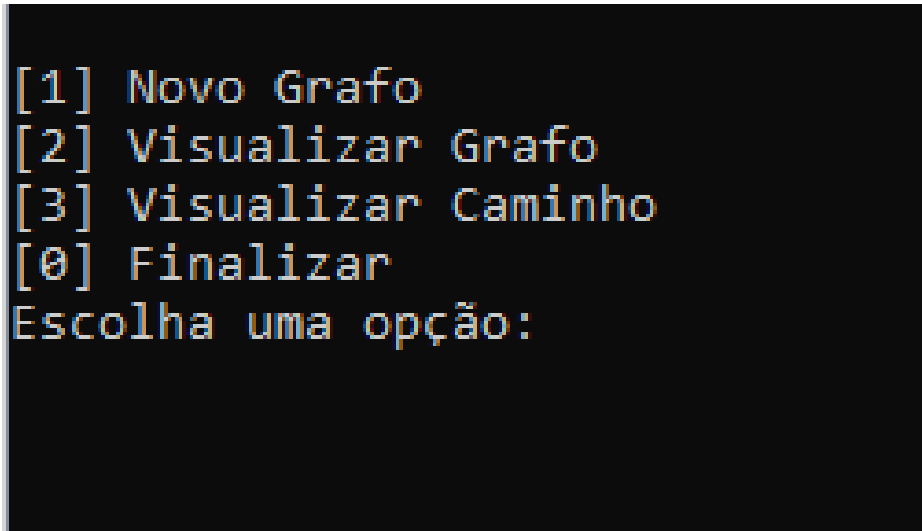
```
int main() {
    setlocale(LC_ALL, "Portuguese");

    int opcao, origem, destino;
    Grafo *grafo = NULL;
    int i, j; // Variáveis de controle
```

### Menu de Opções:

- O usuário é apresentado com um menu de opções numeradas (Fig. 1).
- Ele pode escolher entre as seguintes opções:
- Criar um novo grafo.
- Visualizar o grafo.
- Visualizar o caminho entre vértices.
- Finalizar o programa.
- O usuário é solicitado a escolher uma opção através da entrada padrão usando scanf().

```
do {
    // Menu de opções
    printf("\n[1] Novo Grafo\n[2] Visualizar Grafo\n[3] Visualizar Caminho\n[0]
Finalizar\n");
    printf("Escolha uma opção: ");
    scanf("%d", &opcao);
```



```
[1] Novo Grafo
[2] Visualizar Grafo
[3] Visualizar Caminho
[0] Finalizar
Escolha uma opção:
```

Fig. 1 – Menu inicial

### Switch-case:

- O programa determina a ação a ser tomada com base na opção selecionada pelo usuário.
- Cada caso do switch corresponde a uma opção do menu.
- Se a opção escolhida for 1, o programa cria um novo grafo.
- Se a opção escolhida for 2, o programa visualiza o grafo, se existir.
- Se a opção escolhida for 3, o programa solicita a entrada de vértices de origem e destino e verifica se há um caminho entre eles no grafo.
- Se a opção escolhida for 0, o programa finaliza, liberando a memória alocada para o grafo, se existir.
- Se a opção não corresponder a nenhum dos casos anteriores, o programa exibe uma mensagem de erro e solicita ao usuário que tente novamente.

```
switch (opcao) {
    case 1:
        // Criar um novo grafo
        if (grafo != NULL) {
            liberarGrafo(grafo); // Libera o grafo anterior se existir
        }
        printf("\nInsira o número de vértices: \n");
        scanf("%d", &num_vertices); // Captura o número de vértices
        grafo = criarGrafo(num_vertices); // Cria o grafo com o número especificado
        printf("\nInsira as conexões entre os vértices (1 para conectado, 0 para não
conectado):\n");
        for (i = 0; i < num_vertices; i++) {
            for (j = i; j < num_vertices; j++) {
                if (i == j) {
                    printf("%d para %d: ", i + 1, j + 1);
                    scanf("%d", &grafo->adj_matrix[i][j]);
                } else {
                    printf("%d para %d: ", i + 1, j + 1);
                    scanf("%d", &grafo->adj_matrix[i][j]);
                    grafo->adj_matrix[j][i] = grafo->adj_matrix[i][j]; // A matriz é simétrica
                }
            }
        }
        break;
    case 2:
        // Visualizar o grafo
        if (grafo != NULL) {
            system("cls || clear"); // Limpa a tela
            imprimirGrafo(grafo);
        } else {
            system("cls || clear");
            printf("!!!Grafo não foi criado ainda!!!\n");
        }
        break;
    case 3:
        // Visualizar caminho entre vértices
        if (grafo != NULL) {
            system("cls || clear");
            printf("\nInsira o vértice de origem: ");
```

```

scanf("%d", &origem);
printf("Insira o vértice de destino: ");
scanf("%d", &destino);

// Verificar se há um caminho entre origem e destino
int *visitados = (int *)calloc(grafo->num_vertices, sizeof(int));
int *caminho = (int *)calloc(grafo->num_vertices, sizeof(int));
if (temCaminho(grafo, origem - 1, destino - 1, visitados,
caminho, 0)) {

    system("cls || clear");
    printf("Existe um caminho entre o vértice %d e o vértice
%d.\n", origem, destino);

    printf("\nCaminho: \n");
    printf("%d", origem);
    int i;
    for (i = 1; caminho[i] != destino - 1; i++) {
        printf(" -> %d", caminho[i] + 1);
    }
    printf(" -> %d\n", destino);
} else {
    // Se não houver caminho ou os vértices estiverem fora do
intervalo válido
    if (origem < 1 || origem > grafo->num_vertices || destino < 1 ||
destino > grafo->num_vertices) {
        printf("\nNão há caminho entre o vértice.\n");
    } else {
        printf("\nNão há caminho entre o vértice %d e o vértice
%d.\n", origem, destino);
    }
}
free(visitados);
free(caminho);
} else {
    system("cls || clear");
    printf("\n!!!Grafo não foi criado ainda!!!\n");
}
break;

case 0:
    // Finalizar o programa
    if (grafo != NULL) {
        liberarGrafo(grafo); // Libera a memória alocada para o grafo

    }
    system("cls || clear");
    printf("\nPrograma finalizado.\n");
    break;
default:
    system("cls || clear");
    printf("\nOpção inválida. Tente novamente.\n");
}
} while (opcao != 0);

return 0;
}

```

### Criar um Novo Grafo (Opção 1):

- Libera a memória alocada para o grafo anterior, se existir.
- Solicita ao usuário que insira o número de vértices para o novo grafo (Fig. 2).
- Cria o grafo com o número especificado de vértices.
- Solicita ao usuário que insira as conexões entre os vértices.
- Armazena os valores na matriz de adjacência do grafo.
- Se a matriz não for simétrica, a torna simétrica, pois representa um grafo não direcionado.

```
case 1:
    // Criar um novo grafo
    if (grafo != NULL) {
        liberarGrafo(grafo); // Libera o grafo anterior se existir
    }
    printf("\nInsira o número de vértices: \n");
    scanf("%d", &num_vertices); // Captura o número de vértices
    grafo = criarGrafo(num_vertices); // Cria o grafo com o número especificado
    printf("\nInsira as conexões entre os vértices (1 para conectado, 0 para não
conectado):\n");
    for (i = 0; i < num_vertices; i++) {
        for (j = i; j < num_vertices; j++) {
            if (i == j) {
                printf("%d para %d: ", i + 1, j + 1);
                scanf("%d", &grafo->adj_matrix[i][j]);
            } else {
                printf("%d para %d: ", i + 1, j + 1);
                scanf("%d", &grafo->adj_matrix[i][j]);
                grafo->adj_matrix[j][i] = grafo->adj_matrix[i][j]; // A matriz é simétrica
            }
        }
    }
    break;
```

```

[1] Novo Grafo
[2] Visualizar Grafo
[3] Visualizar Caminho
[0] Finalizar
Escolha uma opção: 1

Insira o número de vértices:
7

Insira as conexões entre os vértices (1 para conectado, 0 para não conectado):
1 para 1: 0
1 para 2: 1
1 para 3: 0
1 para 4: 1
1 para 5: 0
1 para 6: 0
1 para 7: 0
2 para 2: 0
2 para 3: 1
2 para 4: 0
2 para 5: 0
2 para 6: 0
2 para 7: 0
3 para 3: 0
3 para 4: 0
3 para 5: 0
3 para 6: 0
3 para 7: 1
4 para 4: 0
4 para 5: 1
4 para 6: 0
4 para 7: 0
5 para 5: 0
5 para 6: 1
5 para 7: 0
6 para 6: 0
6 para 7: 0
7 para 7: 0

```

**Fig. 2 – Execução da opção [1] “Novo Grafo”  
Entrando com a quantidade de Vértices e ligações dos mesmos**

### Visualizar o Grafo (Opção 2):

- Verifica se o grafo foi criado.
- Limpa a tela do console.
- Imprime a matriz de adjacência do grafo para visualização pelo usuário (Fig. 3).
- Se o grafo não existir, exibe uma mensagem informando que o grafo não foi criado (Fig. 3.1).

```

case 2:
    // Visualizar o grafo
    if (grafo != NULL) {
        system("cls || clear"); // Limpa a tela
        imprimirGrafo(grafo);
    } else {
        system("cls || clear");
        printf("!!!Grafo não foi criado ainda!!!\n");
    }
    break;

```

```

Matriz de Adjacência:
0 1 0 1 0 0 0
1 0 1 0 0 0 0
0 1 0 0 0 0 1
1 0 0 0 1 0 0
0 0 0 1 0 1 0
0 0 0 0 1 0 0
0 0 1 0 0 0 0

[1] Novo Grafo
[2] Visualizar Grafo
[3] Visualizar Caminho
[0] Finalizar
Escolha uma opção:

```

Fig. 3 – Execução da opção [2] – “Visualizar Grafo”

```

!!!Grafo não foi criado ainda!!!

[1] Novo Grafo
[2] Visualizar Grafo
[3] Visualizar Caminho
[0] Finalizar
Escolha uma opção: 1

```

Fig. 3.1 – Execução da opção [2] – “Visualizar Grafo” sem a criação do grafo

### Visualizar Caminho entre Vértices (Opção 3):

- Verifica se o grafo foi criado.
- Limpa a tela do console.
- Solicita ao usuário que insira os vértices de origem e destino (Fig. 4).
- Verifica se há um caminho entre os vértices usando a função temCaminho().
- Se houver um caminho, imprime o caminho na tela (Fig. 4.1).
- Se não houver caminho ou os vértices estiverem fora do intervalo válido, exibe uma mensagem correspondente.
- Libera a memória alocada para os vetores visitados e caminho.

case 3:

```

// Visualizar caminho entre vértices
if (grafo != NULL) {
    system("cls || clear");
    printf("\nInsira o vértice de origem: ");
    scanf("%d", &origem);
    printf("Insira o vértice de destino: ");
    scanf("%d", &destino);

    // Verificar se há um caminho entre origem e destino
    int *visitados = (int *)calloc(grafo->num_vertices, sizeof(int));
    int *caminho = (int *)calloc(grafo->num_vertices, sizeof(int));
    if (temCaminho(grafo, origem - 1, destino - 1, visitados,
caminho, 0)) {

        system("cls || clear");
        printf("Existe um caminho entre o vértice %d e o vértice
%d.\n", origem, destino);

        printf("\nCaminho: \n");
        printf("%d", origem);
        int i;
        for (i = 1; caminho[i] != destino - 1; i++) {
            printf(" -> %d", caminho[i] + 1);
        }
    }
}

```

```

        printf(" -> %d\n", destino);
    } else {
        // Se não houver caminho ou os vértices estiverem fora do
intervalo válido
        if (origem < 1 || origem > grafo->num_vertices || destino < 1 ||
destino > grafo->num_vertices) {
            printf("\nNão há caminho entre o vértice.\n");
        } else {
            printf("\nNão há caminho entre o vértice %d e o vértice
%d.\n", origem, destino);
        }
    }
    free(visitados);
    free(caminho);
} else {
    system("cls || clear");
    printf("\n!!!!Grafo não foi criado ainda!!!\n");
}
break;

```

```

Insira o vértice de origem: 7
Insira o vértice de destino: 6

```

**Fig. 4 – Execução da opção [3] – “Visualizar Caminho”  
Entrando com os valores de Origem e Destino**

```

Existe um caminho entre o vértice 7 e o vértice 6.

Caminho:
7 -> 3 -> 2 -> 1 -> 4 -> 5 -> 6

[1] Novo Grafo
[2] Visualizar Grafo
[3] Visualizar Caminho
[0] Finalizar
Escolha uma opção: _

```

**Fig. 4.1 – Execução da opção [3] – “Visualizar Caminho”  
Resultado**

### Finalizar o Programa (Opção 0):

- Verifica se o grafo foi criado.
- Libera a memória alocada para o grafo.
- Limpa a tela do console.
- Exibe uma mensagem indicando que o programa foi finalizado (Fig. 5).

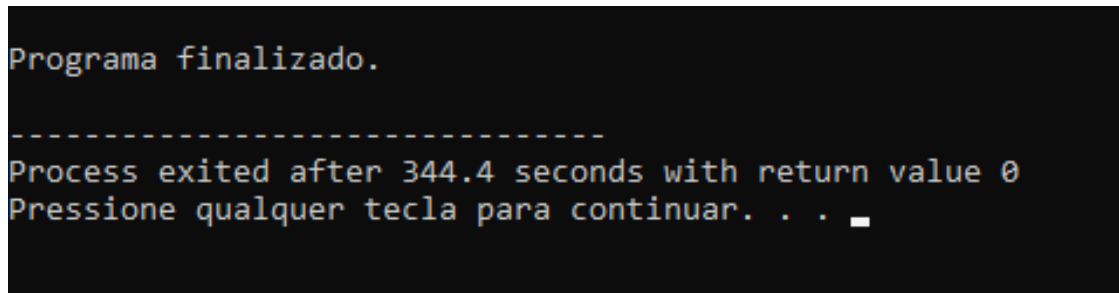
Opção Inválida:

- Se o usuário inserir uma opção inválida, o programa exibe uma mensagem de erro e solicita que ele tente novamente.
- Loop do-while:

O programa continua a executar até que o usuário selecione a opção de finalizar o programa (0).

```
case 0:
    // Finalizar o programa
    if (grafo != NULL) {
        liberarGrafo(grafo); // Libera a memória alocada para o grafo

    }
    system("cls || clear");
    printf("\nPrograma finalizado.\n");
    break;
```



Programa finalizado.

-----

Process exited after 344.4 seconds with return value 0

Pressione qualquer tecla para continuar. . . █

Fig. 5 – Execução da opção [0] “Finalizar”

### 3. Conclusão

O programa oferece uma implementação básica para manipulação de grafos por meio de uma matriz de adjacência em C (Fig. 3). Ele fornece funcionalidades essenciais, como a criação de grafos (Fig. 4.1), adição de arestas (Fig. 2), visualização da estrutura do grafo, verificação de caminhos entre vértices e liberação de memória alocada. Este código pode ser estendido e aprimorado para lidar com operações mais avançadas e tipos diferentes de grafos.



**Aprendizagem Obtida:**

Durante o desenvolvimento deste projeto de manipulação de grafos em linguagem C, foram adquiridos uma série de aprendizados significativos. Primeiramente, compreendeu-se a importância da utilização de estruturas de dados adequadas para representar grafos de forma eficiente, bem como a aplicação prática de conceitos como alocação dinâmica de memória para gerenciar essas estruturas de forma dinâmica.

Explorou-se também a técnica de busca em profundidade (DFS), fundamental para resolver problemas relacionados a grafos, como encontrar caminhos entre vértices. Esta técnica proporcionou insights valiosos sobre como navegar por um grafo de maneira sistemática e eficiente, abrindo caminho para soluções elegantes e eficazes para diversos problemas.

A implementação do código permitiu a prática de conceitos de modularização e organização de código, dividindo as funcionalidades do programa em funções específicas para cada operação sobre o grafo. Além disso, a interação com o usuário por meio de um menu interativo proporcionou uma experiência mais amigável e intuitiva.

Durante o desenvolvimento, foram enfrentados desafios relacionados à lógica de programação para manipulação adequada da matriz de adjacência, bem como na implementação correta da busca em profundidade para verificar a existência de caminhos entre vértices. Esses desafios estimularam a busca por soluções criativas e o aprimoramento das habilidades de resolução de problemas.

Em suma, este projeto proporcionou uma valiosa oportunidade de aplicar os conhecimentos teóricos adquiridos em sala de aula na prática, desenvolvendo habilidades essenciais de programação em C, resolução de problemas e trabalho com estruturas de dados, além de familiarizar-se com o uso de técnicas específicas, como a busca em profundidade, em contextos reais de desenvolvimento de software.

## Referências:

ALGORITMOS. Alocação dinâmica: introdução a malloc e calloc. YouTube, 17 de agosto de 2020. Disponível em: <<https://www.youtube.com/watch?v=reV9kQVLt0>>. Acesso em: 06/04/2024

UNIVERSIDADE FEDERAL DO PARANÁ. 32 Alocação dinâmica de memória. Disponível em: <[https://www.inf.ufpr.br/nicolui/Docs/Livros/C/ArmandoDelgado/notas-32\\_Aloca\\_c\\_cao\\_dinamica\\_mem.html](https://www.inf.ufpr.br/nicolui/Docs/Livros/C/ArmandoDelgado/notas-32_Aloca_c_cao_dinamica_mem.html)>. Acesso em: 06/04/2024.

OPENAI. ChatGPT. 2021. Disponível em: <<https://chat.openai.com/>>. Acesso em: 05 de Abril de 2024. Perguntas: “Faça correções ortográficas no texto a seguir”, “Memória dinâmica, melhor forma de usar”, “Algoritmo de Prim e Kruskal” e “Tipos de Buscas em Grafos”.

Ratusznei, J. (2024). Teoria dos Grafos Aula 04 - Aplicações - Ciclos em grafos e digrafos; - Florestas e árvores; - Grafos bipartidos e ciclos ímpares; - Pontes em grafos; - Articulações em grafos.

Ratusznei, J. (2024). Teoria dos Grafos Aula 06 - Árvores Geradoras de Custos Mínimos - Árvores geradoras de grafos, de custo mínimo, MST, Algoritmos de Prim e de Kruskal e Borukva.

Ratusznei, J. (2023). Estrutura de Dados II Aula 03 - Alocação Dinâmica de Memória.

YES+. Aula 08 - Representando um Grafo em C (Parte 2/2). [S. l.], 19 ago. 2020. 1 vídeo (21min 13s). Publicado pelo usuário [Nome do Usuário]. Disponível em: <<https://www.youtube.com/watch?v=2-yVWulTJJc&t=2s>>. Acesso em: 03 abr. 2024.

YES+. Aula 07 - Representando um Grafo em C (Parte 1/2). [S. l.], 19 ago. 2020. 1 vídeo (21min 26s). Publicado pelo usuário [Nome do Usuário]. Disponível em: <<https://www.youtube.com/watch?v=0Z9dlj6yFXM&t=4s>>. Acesso em: 03 abr. 2024.