UNIVERSIDADE FEDERAL DE VIÇOSA - CAMPUS FLORESTAL

# Trabalho Prático 1 de Teoria e Modelo de Grafos

**Nome:** Aline Cristina Santos Silva
Gustavo Luca Ribeiro Da Silva
Luana Tavares Anselmo
Gabriel Ryan dos Santos Oliveira

**Matrícula:** 5791, 5787, 5364, 4688

Florestal - MG
2024

UFV

# Sumário

UFV

## 1. Introdução

Este trabalho foi desenvolvido para a disciplina de Teoria e Modelo de Grafos – CCF-331, do curso de Ciência da Computação da Universidade Federal de Viçosa, Campus Florestal, com o objetivo de projetar e implementar uma biblioteca em linguagem C para a manipulação de grafos não direcionados e ponderados. A biblioteca foi concebida para atender a uma ampla gama de funcionalidades, desde operações básicas, como o cálculo da ordem, tamanho e densidade do grafo, até funcionalidades mais avançadas, como a detecção de articulações, cálculo de componentes conexas e determinação de caminhos mínimos. O desenvolvimento teve como diretriz a modularidade e a reutilização, permitindo que a biblioteca seja facilmente integrada a outros sistemas e utilizada em diversos contextos computacionais.

A escolha da linguagem C foi motivada tanto pela sua eficiência quanto pela familiaridade dos integrantes do grupo com essa tecnologia, o que possibilitou o desenvolvimento de uma solução que alia desempenho e flexibilidade. Durante o processo de implementação, buscou-se equilibrar clareza, eficiência e facilidade de uso, garantindo que a biblioteca não apenas atenda às demandas do trabalho acadêmico, mas também seja prática e acessível para futuros desenvolvedores. Essa abordagem reforça a relevância do projeto, permitindo que a biblioteca seja aplicada em diferentes cenários que exijam manipulação eficiente de grafos, ampliando seu alcance e utilidade.

## 2. Compilação e Organização

O projeto está organizado de maneira estruturada, com as seguintes pastas e arquivos:

**source**: Contém os arquivos-fonte responsáveis pela implementação das funções que compõem a biblioteca.

**headers**: Inclui os arquivos de cabeçalho que definem as funções e estruturas de dados utilizadas no projeto, garantindo modularidade e clareza.

**main**: Contém o arquivo principal do programa, responsável por testar as funcionalidades da biblioteca e interagir com o usuário.

**entradaPadrao.txt**: Arquivo de texto contendo o grafo fornecido na especificação do trabalho, utilizado como entrada padrão para os testes.

**MakeFile**: Arquivo que automatiza o processo de compilação e execução do programa, simplificando a interação com o projeto.

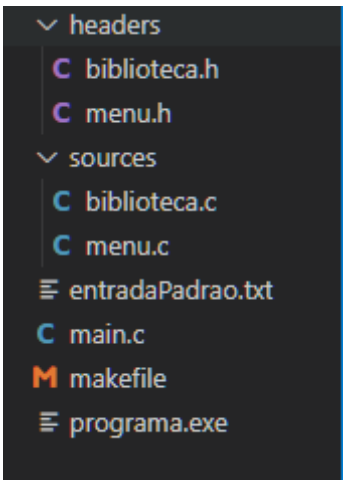


**Figura 1: Estrutura do Projeto**

Para facilitar o processo de compilação, foi criado um Makefile, que oferece comandos simples e eficientes para compilar e executar o programa. Os principais comandos disponíveis são:
**make** ou **make compile**: Compila o projeto, gerando os arquivos executáveis.
**make run**: Executa o programa principal, permitindo testar as funcionalidades implementadas na biblioteca.
**make all**: Realiza o processo completo, compilando o projeto e em seguida executando o programa principal.

```
1  compile:
2      gcc main.c sources/biblioteca.c sources/menu.c -lm -o programa
3
4  run:
5      ./programa
6
7  all: compile run
```

**Figura 2: Comandos Makefile**

Essa organização garante que o projeto seja de fácil manutenção, permitindo que novos desenvolvedores compreendam sua estrutura rapidamente. Além disso, o uso do Makefile automatiza tarefas repetitivas, otimizando o tempo dedicado ao desenvolvimento e à execução dos testes.

## 3. Desenvolvimento

O desenvolvimento deste projeto consistiu na criação de uma biblioteca modular em C para manipulação de grafos não direcionados e ponderados. O processo foi estruturado em etapas claras, começando pela análise dos requisitos, seguida pela implementação das funcionalidades principais e culminando na validação por meio de testes rigorosos.

## 3.1. Representação do Grafo

A estrutura do grafo foi implementada utilizando uma matriz de adjacência ponderada, que foi escolhida por sua simplicidade e eficiência na manipulação de arestas e na consulta de pesos entre vértices. Cada posição da matriz representa o peso da aresta correspondente, enquanto valores especiais indicam a ausência de conexões. Essa escolha foi essencial para a implementação direta e eficiente dos algoritmos desenvolvidos, além de proporcionar uma base sólida para manipulação e análise dos grafos.

## 3.2. Cálculo de Caminhos Mínimos com Bellman-Ford

A função bellman_ford desempenha um papel central no projeto, calculando as menores distâncias entre um vértice inicial e os demais vértices do grafo. O algoritmo é capaz de identificar ciclos negativos, que são informados ao usuário por meio de mensagens de alerta.
Essa verificação é crucial, já que ciclos negativos podem comprometer os resultados do algoritmo, fazendo com que os valores das distâncias mais curtas se tornem infinitamente pequenos. O algoritmo foi otimizado para detectar essas condições de forma eficiente, iterando sobre as arestas do grafo até que as mudanças nas distâncias cessem ou até que a presença de ciclos negativos seja confirmada.

```c
void bellman_ford(Grafo *g, int vertice_inicial)
{
    int n_vertices = g->n_vertices;
    float *distancias = (float *)malloc(n_vertices * sizeof(float));
    int *pai = (int *)malloc(n_vertices * sizeof(int));

    initialize_single_source(g, vertice_inicial, distancias, pai);

    // Relaxa as arestas (n_vertices - 1) vezes
    for (int i = 0; i < n_vertices - 1; i++)
    {
        for (int u = 0; u < n_vertices; u++)
        {
            for (int v = 0; v < n_vertices; v++)
            {
                if (g->matriz_pesos[u][v] != 0.0)
                { // Se há uma aresta entre u e v
                    relax(u, v, g->matriz_pesos[u][v], distancias, pai);
                }
            }
        }
    }

    // Verifica se há ciclos de peso negativo
    bool ciclo_negativo = false;
    for (int u = 0; u < n_vertices; u++)
    {
        for (int v = 0; v < n_vertices; v++)
        {
            if (g->matriz_pesos[u][v] != 0.0)
            { // Se há uma aresta
                if (distancias[u] != FLT_MAX && distancias[u] + g->matriz_pesos
                [u][v] < distancias[v])
                {
                    ciclo_negativo = true; // Ciclo negativo encontrado
                }
            }
        }
    }
```

**Figura 3: Função bellman_ford**

## 3.3. Geração de Grafos Aleatórios

A função gerar_grafo_aleatorio foi criada como uma extensão prática para facilitar testes e validações. Com ela, é possível gerar grafos com pesos positivos, negativos ou mistos, permitindo testar os algoritmos em diferentes cenários. O usuário pode especificar o número de vértices e arestas do grafo, simulando redes reais ou casos de otimização com custos negativos.

Essa funcionalidade também é valiosa para testes de estresse, permitindo criar grafos de diferentes tamanhos e topologias. Quando o grafo contém pesos negativos, a função alerta o usuário, destacando possíveis impactos na execução de algoritmos como Bellman-Ford ou Dijkstra.

```c
void gerar_grafo_aleatorio(int n_vertices, int n_arestas, const char *nome_arquivo, int tipo_peso) {
    FILE *arquivo = fopen(nome_arquivo, "w");
    if (!arquivo) {
        printf("Erro ao criar o arquivo: %s\n", nome_arquivo);
        return;
    }

    fprintf(arquivo, "%d\n", n_vertices); // Escreve o número de vértices

    // Para evitar arestas duplicadas, vamos usar um conjunto
    bool **arestas_existentes = (bool **)malloc(n_vertices * sizeof(bool *));
    for (int i = 0; i < n_vertices; i++) {
        arestas_existentes[i] = (bool *)calloc(n_vertices, sizeof(bool));
    }

    int arestas_criadas = 0;
    while (arestas_criadas < n_arestas) {
        int v1 = rand() % n_vertices; // Vértice 1
        int v2 = rand() % n_vertices; // Vértice 2

        // Evita arestas de um vértice para ele mesmo e arestas duplicadas
        if (v1 != v2 && !arestas_existentes[v1][v2]) {
            float peso;
            if (tipo_peso == 1) {
                peso = ((float)rand() / RAND_MAX) * 10; // Peso positivo entre 0 e 10
            } else if (tipo_peso == 2) {
                peso = ((float)rand() / RAND_MAX) * 10 - 10; // Peso negativo entre -10 e 0
            } else {
                peso = ((float)rand() / RAND_MAX) * 20 - 10; // Peso entre -10 e 10
            }
            fprintf(arquivo, "%d %d %.2f\n", v1 + 1, v2 + 1, peso); // Escreve a aresta
            arestas_existentes[v1][v2] = true; // Marca aresta como existente
            arestas_existentes[v2][v1] = true; // Para grafos não direcionados
            arestas_criadas++;
        }
    }

    // Libera a memória
    for (int i = 0; i < n_vertices; i++) {
        free(arestas_existentes[i]);
    }
    free(arestas_existentes);
    fclose(arquivo);
    printf("Grafo gerado e salvo em %s\n", nome_arquivo);
```

**Figura 4: Função gerar_grafo_aleatorio**

## 3.4. Busca em Profundidade para Componentes Conexas

A função dfs_componente_conexa realiza uma busca em profundidade (DFS) a partir de um vértice não visitado, identificando todos os vértices que pertencem à mesma componente conectada. Ela utiliza vetores auxiliares para rastrear os vértices visitados e armazenar os vértices da componente atual.

Complementando essa funcionalidade, a função componentes_conexas percorre todos os vértices do grafo, iniciando novas buscas para cada vértice não visitado. Ela identifica e conta o número total de componentes conexas, além de listar os vértices de cada componente. Essas funções são fundamentais para a análise estrutural do grafo e são amplamente utilizadas em várias partes do código.

```c
void dfs_componente_conexa(Grafo *g, int vertice, bool *visitado, int *componente, int *tamanho) {
    visitado[vertice] = true;
    componente[*tamanho] = vertice + 1; // Armazena o vértice (ajusta para 1-indexado)
    (*tamanho)++;

    for (int i = 0; i < g->n_vertices; i++) {
        if (g->matriz_pesos[vertice][i] != 0.0 && !visitado[i]) { // Se há uma aresta e não foi visitado
            dfs_componente_conexa(g, i, visitado, componente, tamanho);
        }
    }
}
```

**Figura 5: Função  dfs_componente_conexa**

```c
void componentes_conexas(Grafo *g) {
    bool *visitado = (bool *)malloc(g->n_vertices * sizeof(bool));
    int *componente = (int *)malloc(g->n_vertices * sizeof(int));
    int numero_componentes = 0;

    for (int i = 0; i < g->n_vertices; i++) {
        visitado[i] = false; // Inicializa todos os vértices como não visitados
    }

    printf("Componentes conexas:\n");

    for (int i = 0; i < g->n_vertices; i++) {
        if (!visitado[i]) { // Se o vértice não foi visitado, é o início de uma nova componente
            int tamanho = 0;
            dfs_componente_conexa(g, i, visitado, componente, &tamanho);
            numero_componentes++;

            printf("Componente %d: ", numero_componentes);
            for (int j = 0; j < tamanho; j++) {
                printf("%d ", componente[j]);
            }
            printf("\n");
        }
    }

    printf("Número total de componentes conexas: %d\n", numero_componentes);

    // Libera a memória
    free(visitado);
    free(componente);
}
```

**Figura 6: Função componentes_conexas**

### 3.5. Relevância e Integração

As funções descritas acima são cruciais para o funcionamento da biblioteca. A implementação do algoritmo Bellman-Ford garante a análise de caminhos mínimos mesmo em cenários complexos. A geração de grafos aleatórios permite testar e validar os algoritmos sob diversas condições, enquanto as funções de busca em profundidade fornecem uma análise robusta da conectividade do grafo.
Ao combinar modularidade, eficiência e flexibilidade, o projeto oferece uma solução abrangente para a manipulação e análise de grafos, atendendo aos requisitos da disciplina e estendendo suas capacidades além das especificações iniciais.

## 4. Resultado

Os testes realizados confirmaram a eficácia e precisão da biblioteca desenvolvida para manipulação de grafos. A função bellman_ford foi capaz de calcular corretamente as menores distâncias entre os vértices e identificar ciclos negativos, emitindo alertas apropriados para esses casos. A funcionalidade de geração de grafos aleatórios provou ser valiosa ao criar cenários variados, permitindo testar a performance em topologias complexas e com diferentes configurações de pesos, incluindo valores negativos.
As funções relacionadas à análise de componentes conectadas demonstraram robustez, identificando corretamente agrupamentos no grafo e fornecendo informações detalhadas sobre sua estrutura. Além disso, a organização modular do código facilitou a integração e reutilização das funções, permitindo uma adaptação simples para diferentes contextos.
Os resultados obtidos, aliados à validação por meio de gráficos e testes práticos, evidenciam o potencial da biblioteca como uma ferramenta confiável e flexível para estudo e aplicação em problemas que envolvem grafos não direcionados e ponderados.

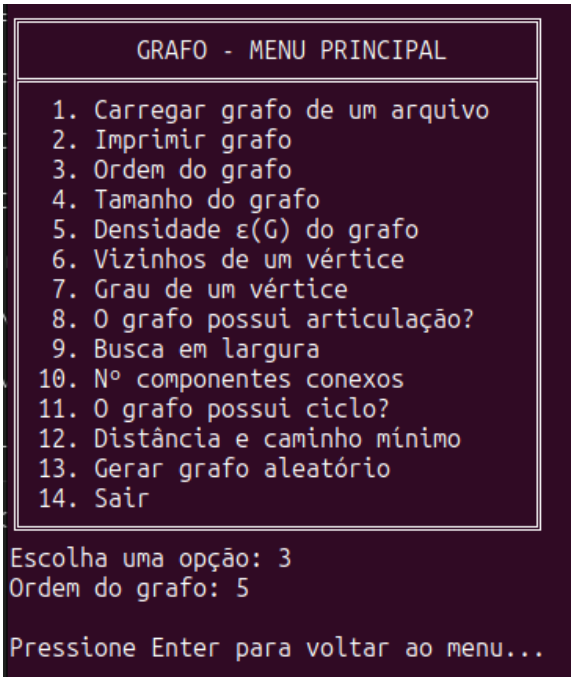


**Figura 7: Ordem do Grafo**

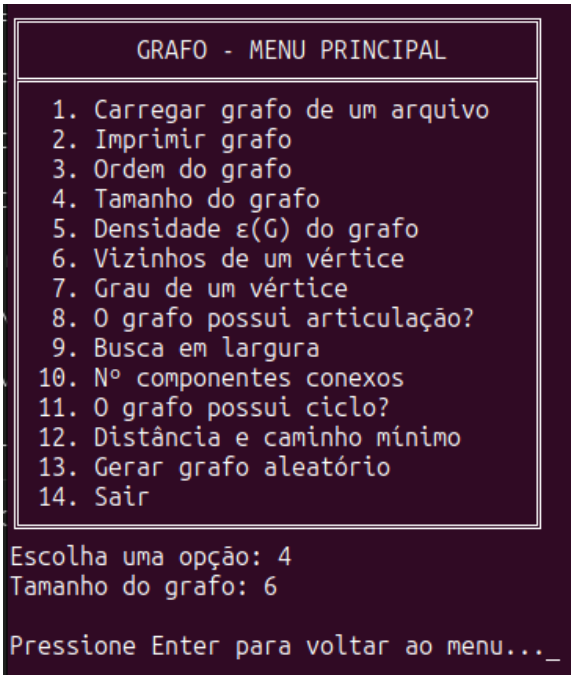


**Figura 8: Tamanho do Grafo**

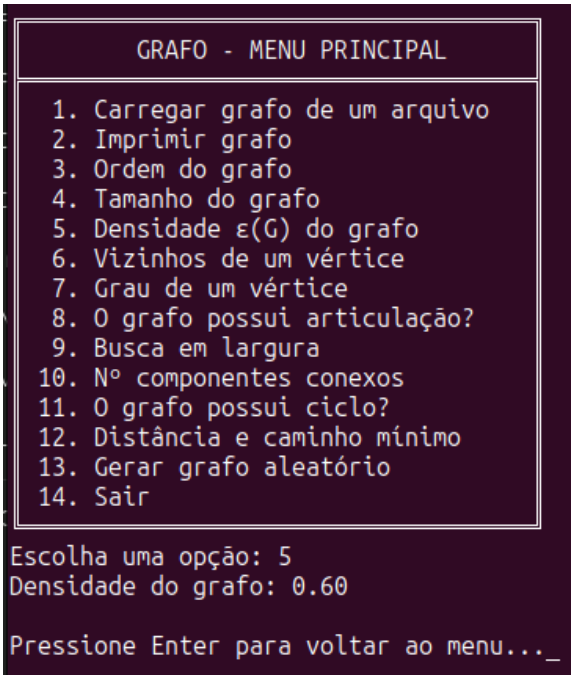


**Figura 9: Densidade ε(G) do Grafo**

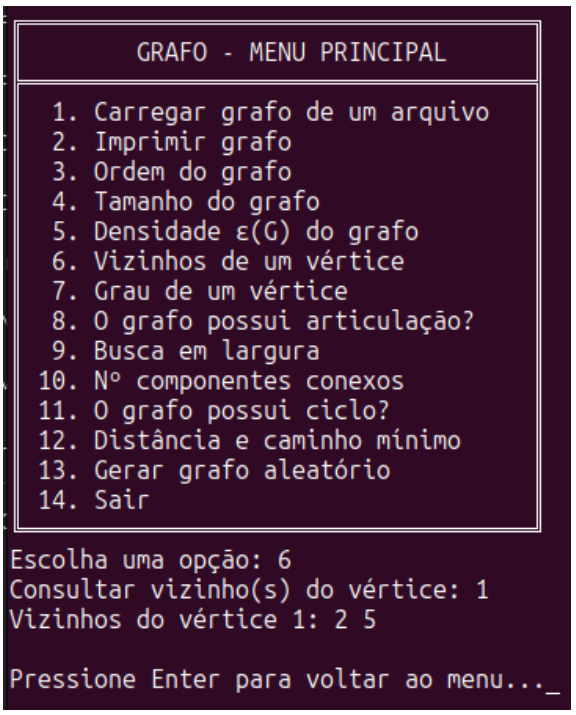


**Figura 10: Vizinhos de um Vértice**

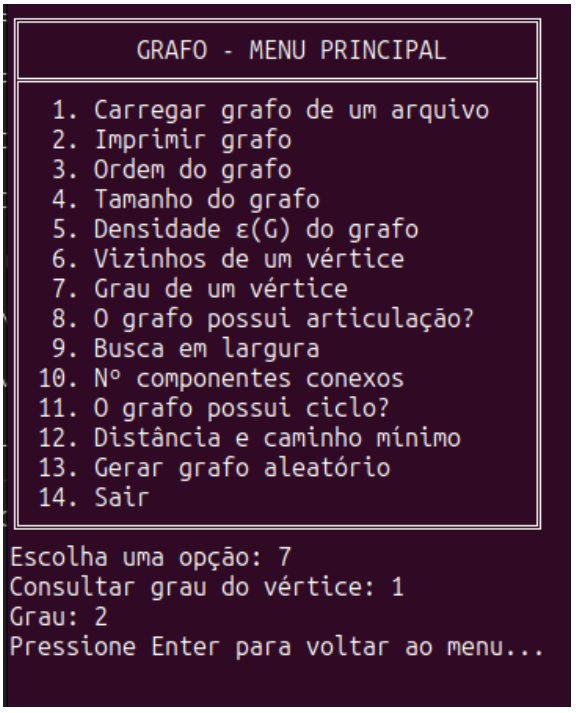


**Figura 11: Grau de um Vértice**

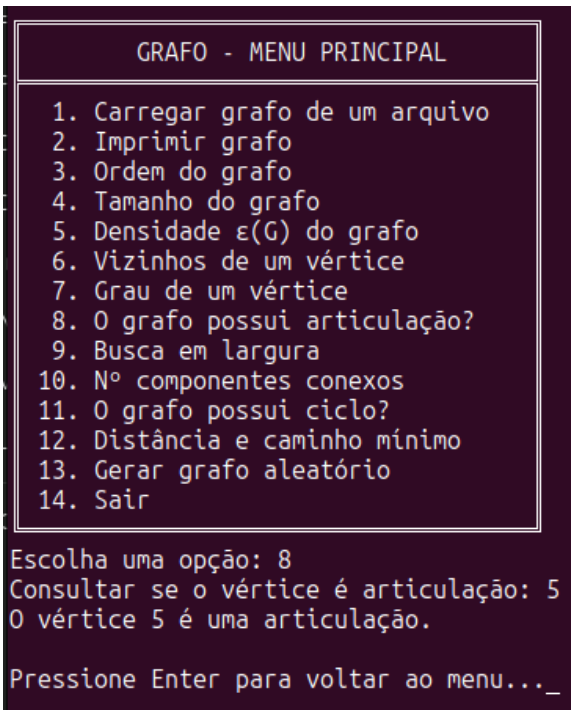


**Figura 12: Verificar se Vértice é Articulação**

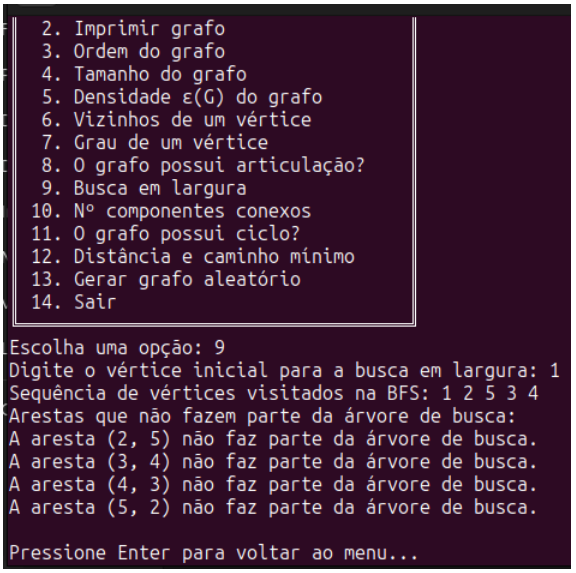


**Figura 13: Busca em Largura**

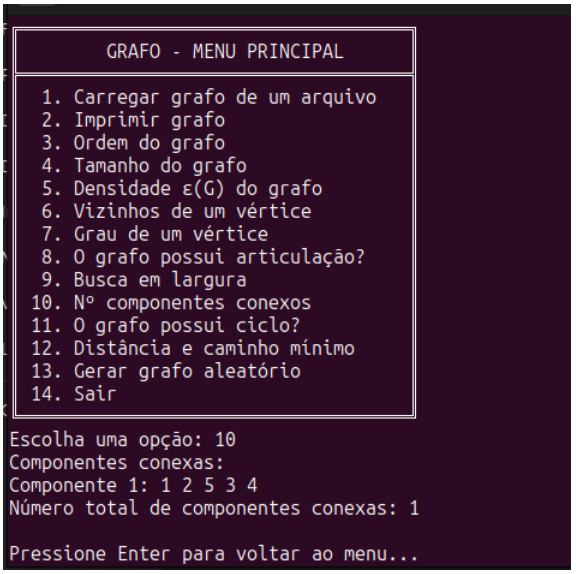


**Figura 13: Nº Componentes Conexos**

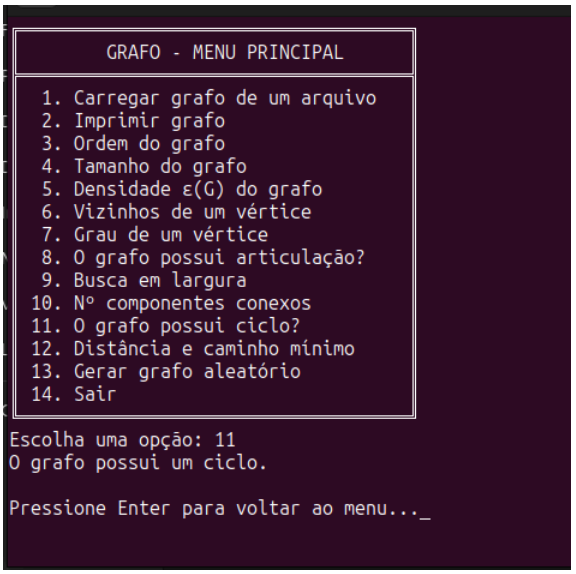


**Figura 14: Verificar se Grafo Possui Ciclos**

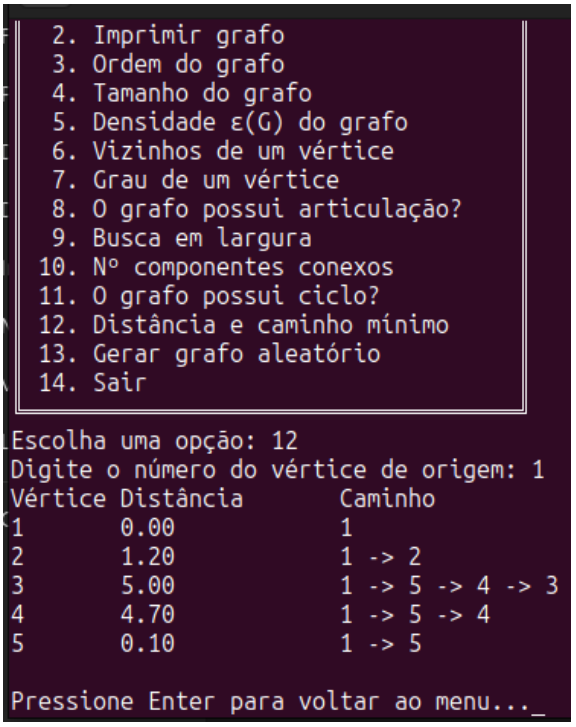


**Figura 15: Distância e Menor Caminho**

## 5. Conclusão

O desenvolvimento deste projeto resultou em uma biblioteca modular e eficiente para a manipulação de grafos não direcionados e ponderados, atendendo aos objetivos propostos e enfrentando desafios significativos, como o tratamento de pesos negativos e a detecção de ciclos negativos. A lógica empregada na função de cálculo da distância e do menor caminho demonstrou-se uma abordagem eficaz para lidar com grafos com ponderação negativa, garantindo resultados confiáveis mesmo em cenários complexos.

A escolha pela matriz de adjacência como estrutura base revelou-se acertada, proporcionando simplicidade e eficiência na manipulação de arestas e na execução dos algoritmos. Funcionalidades adicionais, como a geração de grafos aleatórios, ampliaram as possibilidades de teste e validação, tornando a biblioteca versátil para explorar diferentes comportamentos e cenários.

A implementação dos algoritmos exigiu uma compreensão aprofundada dos conceitos de grafos e das estruturas de dados em C, consolidando conhecimentos teóricos em práticas robustas de programação. A organização modular do código, aliada à documentação detalhada, garante sua reutilização e facilita futuras extensões.

Concluímos que a biblioteca desenvolvida é uma ferramenta valiosa tanto para o estudo quanto para aplicações práticas envolvendo a análise e manipulação de grafos. Além disso, o trabalho reforça a importância de abordagens otimizadas e bem planejadas na resolução de problemas, contribuindo para avanços no campo de algoritmos e estruturas de dados.

## 6. Referência

[1] CORMEN, TH; LEISERSON, E.C.; RIVEST, RL; STEIN, C. Introdução aos Algoritmos . 3ª edição. Rio de Janeiro: Elsevier, 2011.
[2] SEDGEWICK, R.; WAYNE, K. Algoritmos em C: Partes 1-4. 3. ed. São Paulo: Pearson Education, 2011.
[3] TARIAN, RE "Algoritmos de busca em profundidade e de grafos lineares." SIAM Journal on Computing , v. 1, n. 2, p.
[4] Bellman, RE; Ford, LR "Abordagem de programação dinâmica para o problema do caixeiro viajante." Journal of the ACM , v. 4, n. 1, p. 16-26, 1964.