PONTIFÍCIA UNIVERSIDADE CATÓLICA
SISTEMAS DE INFORMAÇÃO

GUSTAVO RODRIGUES
MOISÉS MAIA
OTÁVIO NASCIMENTO
SARAH PALHARES

Algoritmos em Grafos

Trabalho Prático

BETIM-MG
2025

GUSTAVO RODRIGUES
MOISÉS MAIA
OTÁVIO NASCIMENTO
SARAH PALHARES

Algoritmo em Grafos

Trabalho Prático

Professora: Roselene Henrique Pereira Costa

BETIM-MG
2025

Sumário

Introdução

Este trabalho apresenta o desenvolvimento de um Sistema de Análise de Grafos implementado em C#, voltado para a representação e análise de grafos ponderados não direcionados. O sistema foi projetado para realizar operações fundamentais de teoria dos grafos através de algoritmos clássicos de travessia e otimização.

O sistema implementa uma estrutura de dados baseada em matriz de adjacência para representar grafos ponderados não direcionados. A entrada de dados é realizada através da leitura de um arquivo texto contendo a lista de arestas com seus respectivos pesos. Sobre essa estrutura, foram implementados algoritmos essenciais para análise e processamento de grafos.

Os principais objetivos deste trabalho são:

- Implementar uma representação eficiente de grafos ponderados não direcionados utilizando matriz de adjacência
- Desenvolver algoritmos de travessia em grafos (DFS e BFS) com registro de tempos de descoberta e término
- Implementar o algoritmo de Dijkstra para encontrar o menor caminho entre dois vértices
- Desenvolver algoritmo de árvore geradora mínima (Prim ou Kruskal)
- Proporcionar uma interface simples e clara para visualização dos resultados das operações realizadas

O sistema foi desenvolvido com foco em clareza de código, eficiência computacional e facilidade de uso, permitindo a análise prática dos conceitos estudados em teoria dos grafos.

O sistema está hospedado em um repositório no GitHub:
https://github.com/otavio-castro/tp-grafos

Travessia em Profundidade (BFS)

DFS é um algoritmo de travessia em grafos que explora o mais profundo possível antes de retroceder. Imagine como se você estivesse explorando um labirinto sempre indo o mais longe possível em cada caminho antes de voltar. Ele funciona da seguinte maneira:

Começa em um vértice inicial, marca o vértice como visitado, visita recursivamente todos os vizinhos não visitados, retrocede quando não há mais vizinhos para visitar e repete até explorar todos os vértices alcançáveis.

Esse algoritmo tem aplicabilidade prática em diversos contextos, sendo eles detecção de ciclos em grafos, verificar conectividade entre vértices, ordenação topológica, resolver labirintos e puzzles, encontrar componentes conexos e verificar se um grafo é bipartido.

Entre suas vantagens, podemos destacar um uso menor de memória que BFS (travessia em largura), simplicidade de implementação com recursão, e bom para explorar até o fim de um caminho. Porém ele também possui desvantagens, como não garantir o caminho mais curto, poder ficar preso em grafos infinitos e a recursividade pode causar stack overflow em grafos grandes.

- Explicação dos principais trechos de código:

```
using System;
using System.IO;
using Spectre.Console;

3 referências
class Grafo
{
    // Representa um grafo não-direcionado com matriz de adjacência e suporte para DFS.
    private int vertices;
    private int[,] matriz;
    private int tempo;
    private int[] descoberta;
    private int[] termino;

    1 referência
    public Grafo(int v)
    {
        vertices = v;
        matriz = new int[v, v];
    }
```

Esse trecho de código cria a estrutura do grafo com seus atributos (vértices, matriz de conexões, tempo, descoberta e término) e inicializa a matriz vazia no construtor.

```
public void AdicionarAresta(int v, int u, int peso)
{
    matriz[v, u] = peso;
    matriz[u, v] = peso;
}
```

Essa função adiciona uma aresta entre dois vértices com um peso específico. Como o grafo é não-direcionado, adiciona nos dois sentidos.

```
public void DFS(int inicio)
{
    bool[] visitado = new bool[vertices];
    descoberta = new int[vertices];
    termino = new int[vertices];
    tempo = 0;

    AnsiConsole.Write(
        new Rule("[yellow]Árvore DFS[/]")
            .RuleStyle("grey")
            .LeftJustified());

    var tree = new Tree("[cyan]Estrutura da Árvore DFS[/]");
    DFSUtil(inicio, visitado, -1, tree, null);
    AnsiConsole.Write(tree);

    // Tabela de tempos
    var table = new Table()
        .Border(TableBorder.Rounded)
        .BorderColor(Color.Blue);

    table.AddColumn(new TableColumn("[yellow]Vértice[/]").Centered());
    table.AddColumn(new TableColumn("[green]Descoberta[/]").Centered());
    table.AddColumn(new TableColumn("[red]Término[/]").Centered());

    for (int i = 0; i < vertices; i++)
    {
        if (descoberta[i] > 0)
        {
            table.AddRow(
                $"[cyan]{i}[/]",
                $"[green]{descoberta[i]}[/]",
                $"[red]{termino[i]}[/]"
            );
        }
    }

    AnsiConsole.WriteLine();
    AnsiConsole.Write(
        new Rule("[yellow]Tempos de Descoberta e Término[/]")
            .RuleStyle("grey")
            .LeftJustified());
    AnsiConsole.Write(table);
}
```

Essa função dá início a travessia em profundidade (DFS) a partir de um vértice. Ela cria os vetores de controle (visitado, descoberta, término), chama recursivamente

DFSUtil para construir a árvore de exploração e exibe visualmente os resultados usando Spectre.Console com uma árvore hierárquica e uma tabela formatada dos tempos de descoberta e término.

```
private void DFSUtil(int v, bool[] visitado, int pai, Tree tree, TreeNode? parentNode)
{
    visitado[v] = true;
    tempo++;
    descoberta[v] = tempo;

    TreeNode currentNode;
    if (pai == -1)
    {
        currentNode = tree.AddNode($"[bold green]Raiz: {v}[/] [dim][/]");
    }
    else
    {
        currentNode = parentNode!.AddNode($"[cyan]{v}[/] [dim][/]");
    }

    for (int u = 0; u < vertices; u++)
    {
        if (matriz[v, u] > 0 && !visitado[u])
        {
            DFSUtil(u, visitado, v, tree, currentNode);
        }
    }

    tempo++;
    termino[v] = tempo;
}
```

Essa função faz a travessia em profundidade recursivamente. Ela marca o vértice como visitado, registra o tempo de descoberta, adiciona o vértice na árvore visual (como raiz ou filho), explora recursivamente todos os vizinhos não visitados e quando finalizar, registra o tempo de término.

```
public void ImprimirArestas()
{
    var table = new Table()
        .Border(TableBorder.Double)
        .BorderColor(Color.Green);

    table.AddColumn(new TableColumn("[yellow]Vértice U[/]").Centered());
    table.AddColumn(new TableColumn("[yellow]Vértice V[/]").Centered());
    table.AddColumn(new TableColumn("[yellow]Peso[/]").Centered());

    for (int i = 0; i < vertices; i++)
    {
        for (int j = i + 1; j < vertices; j++)
        {
            if (matriz[i, j] > 0)
            {
                table.AddRow(
                    $"[cyan]{i}[/]",
                    $"[cyan]{j}[/]",
                    $"[green]{matriz[i, j]}[/]"
                );
            }
        }
    }

    AnsiConsole.Write(
        new Rule("[bold blue]Arestas do Grafo[/]")
            .RuleStyle("grey")
            .LeftJustified());
    AnsiConsole.Write(table);
}
```

Essa função imprime todas as arestas do grafo com seus pesos. Ela cria e exibe uma tabela formatada com todas as arestas do grafo, mostrando os pares de vértices conectados com seus respectivos pesos, com um cabeçalho visual.

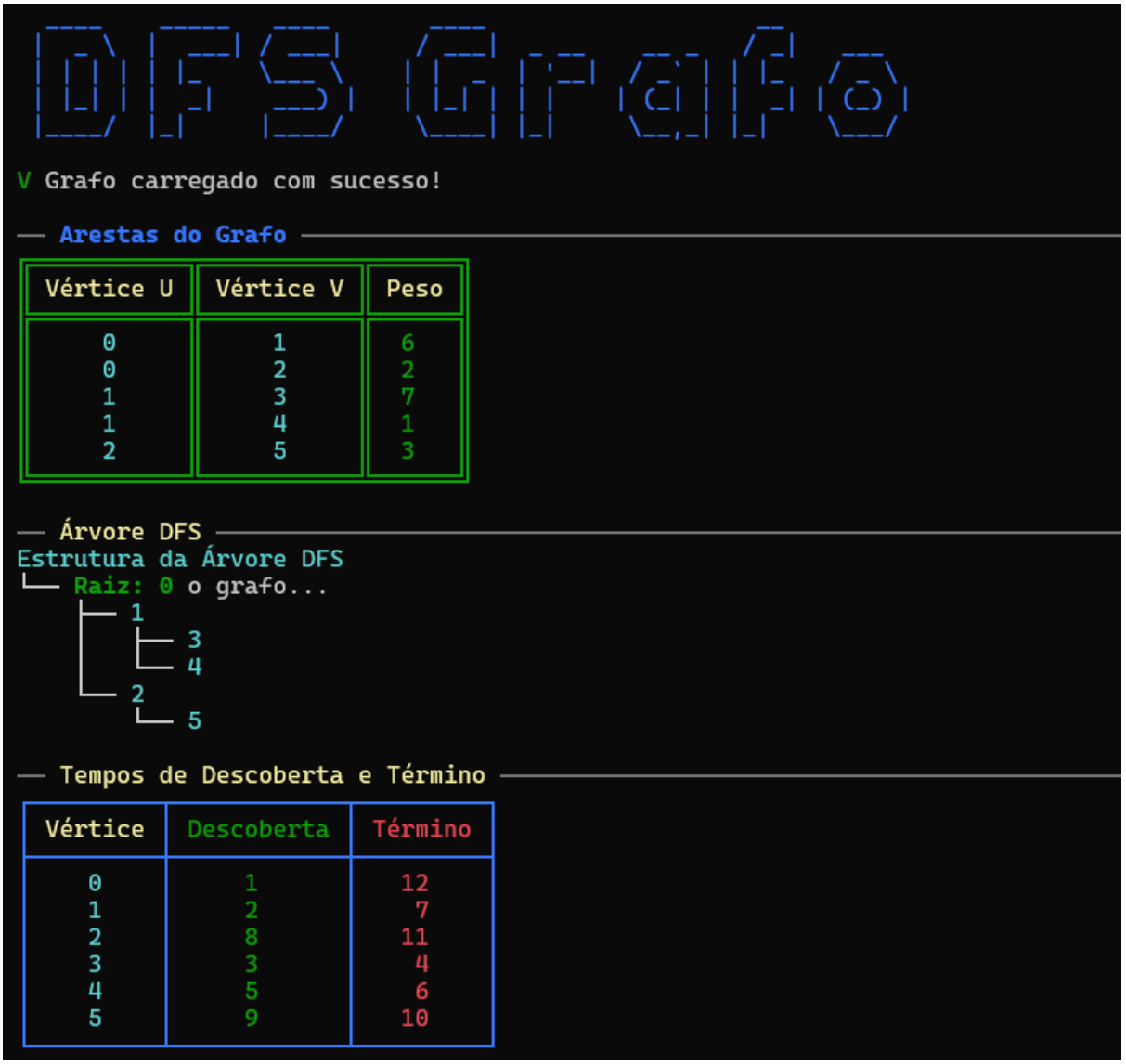
```
class Program
{
    // Carrega grafo do arquivo, executa DFS e exibe resultados com interface visual animada.

    1 referência
    static string CaminhoDoProjeto(string arquivo)
    {
        string dir = Directory.GetCurrentDirectory();

        while (dir != null)
        {
            var csproj = Directory.GetFiles(dir, "*.csproj");

            if (csproj.Length > 0)
                return Path.Combine(dir, arquivo);

            dir = Directory.GetParent(dir)?.FullName;
        }

        throw new Exception($"Arquivo '{arquivo}' não encontrado no projeto");
    }
```

A função sobe diretórios a partir da pasta atual até encontrar um arquivo .csproj (a raiz do projeto) e quando encontra, retorna o caminho completo do arquivo passado como parâmetro dentro dessa pasta. Se não encontrar, lança exceção.

```
static void Main(string[] args)
{
    try
    {
        AnsiConsole.Write(
            new FigletText("DFS Grafo")
                .LeftJustified()
                .Color(Color.Blue));

        AnsiConsole.Status()
            .Start("Carregando grafo...", ctx =>
            {
                ctx.Spinner(Spinner.Known.Star);
                ctx.SpinnerStyle(Style.Parse("green"));

                string caminho = CaminhoDoProjeto("grafo.txt");
                string[] linhas = File.ReadAllLines(caminho);
                int numVertices = int.Parse(linhas[0]);

                ctx.Status($"Criando grafo com {numVertices} vértices...");
                Grafo g = new Grafo(numVertices);

                ctx.Status("Adicionando arestas...");
                for (int i = 1; i < linhas.Length; i++)
                {
                    string[] partes = linhas[i].Split(' ');
                    int v = int.Parse(partes[0]);
                    int u = int.Parse(partes[1]);
                    int peso = int.Parse(partes[2]);
                    g.AdicionarAresta(v, u, peso);
                }

                ctx.Status("Processando grafo...");
                System.Threading.Thread.Sleep(500);

                AnsiConsole.MarkupLine("[green]√[/] Grafo carregado com sucesso!");
                AnsiConsole.WriteLine();

                g.ImprimirArestas();
                AnsiConsole.WriteLine();
                g.DFS(0);
            });

        AnsiConsole.WriteLine();
        AnsiConsole.Markup("[dim]Pressione qualquer tecla para sair...[/]");
        Console.ReadKey();
    }
    catch (Exception ex)
    {
        AnsiConsole.WriteException(ex);
    }
}
```

Programa principal que lê o arquivo grafo.txt, cria um grafo com o número de vértices da primeira linha, adiciona todas as arestas das demais linhas, imprime as arestas e executa o DFS a partir do vértice 0.

- Resultados obtidos:



A execução do programa mostrou como o algoritmo de busca em profundidade percorreu o grafo. Primeiro, foram listadas todas as conexões entre os vértices junto com seus pesos. Em seguida, a árvore DFS revelou que a exploração começou pelo vértice 0 e seguiu visitando primeiro todos os vértices alcançáveis através do vértice 1, depois retornou e explorou os vértices alcançáveis através do vértice 2. Os tempos registrados mostraram que o vértice inicial levou 12 unidades de tempo para completar toda a exploração, enquanto os vértices que não tinham mais conexões para explorar foram finalizados rapidamente, evidenciando a característica do algoritmo de ir o mais fundo possível em cada caminho antes de voltar e tentar outro.

Travessia em Amplitude (BFS)

Executando o Programa

Passo 1 – Compilação e Execução

Para executar o sistema, o usuário deve acessar a pasta do projeto e utilizar o comando:

cd amplitude

dotnet run

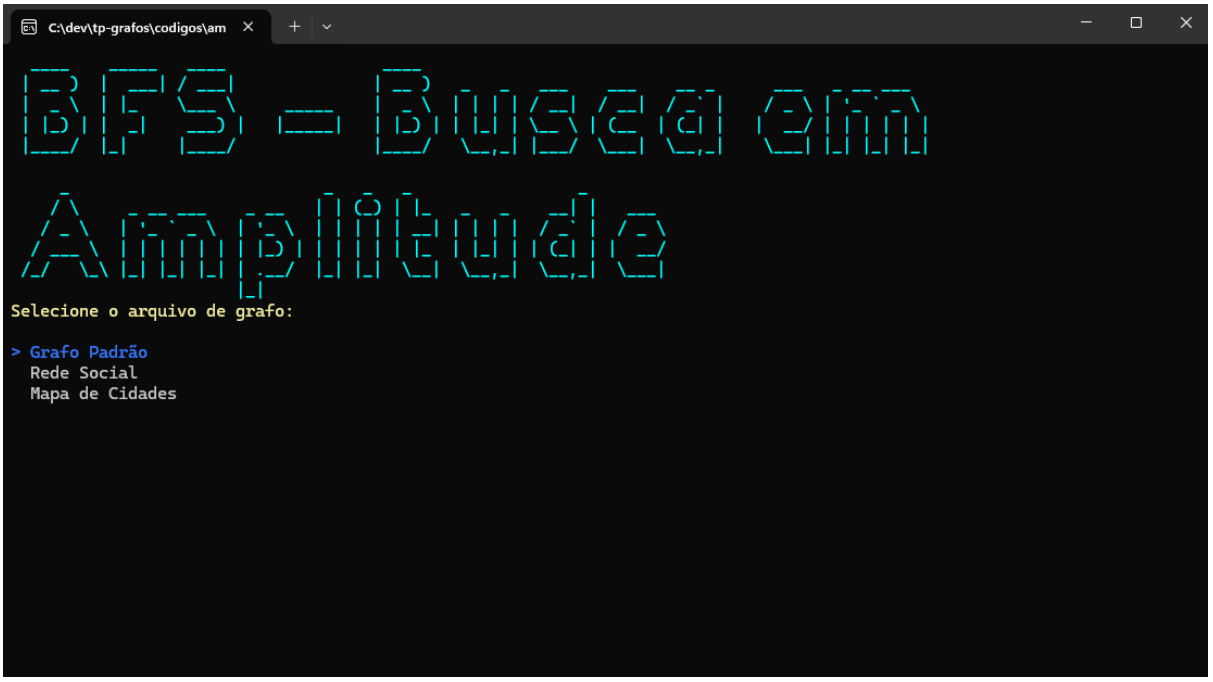
Passo 2 – Seleção do Grafo

Ao iniciar o programa, é exibido um menu para que o usuário escolha qual grafo deseja carregar. As opções incluem:

- Grafo Padrão

- Rede Social

- Mapa de Cidades



Após a seleção, o programa apresenta a estrutura do grafo, listando cada vértice acompanhado de suas adjacências.

O usuário informa um vértice inicial para executar a busca em amplitude (BFS).

O programa mostra:

1. **Passos da execução da BFS**
2. **Ordem de visitação dos vértices**
3. **Níveis de distância (camadas) a partir do vértice inicial**

Exemplos de Grafos

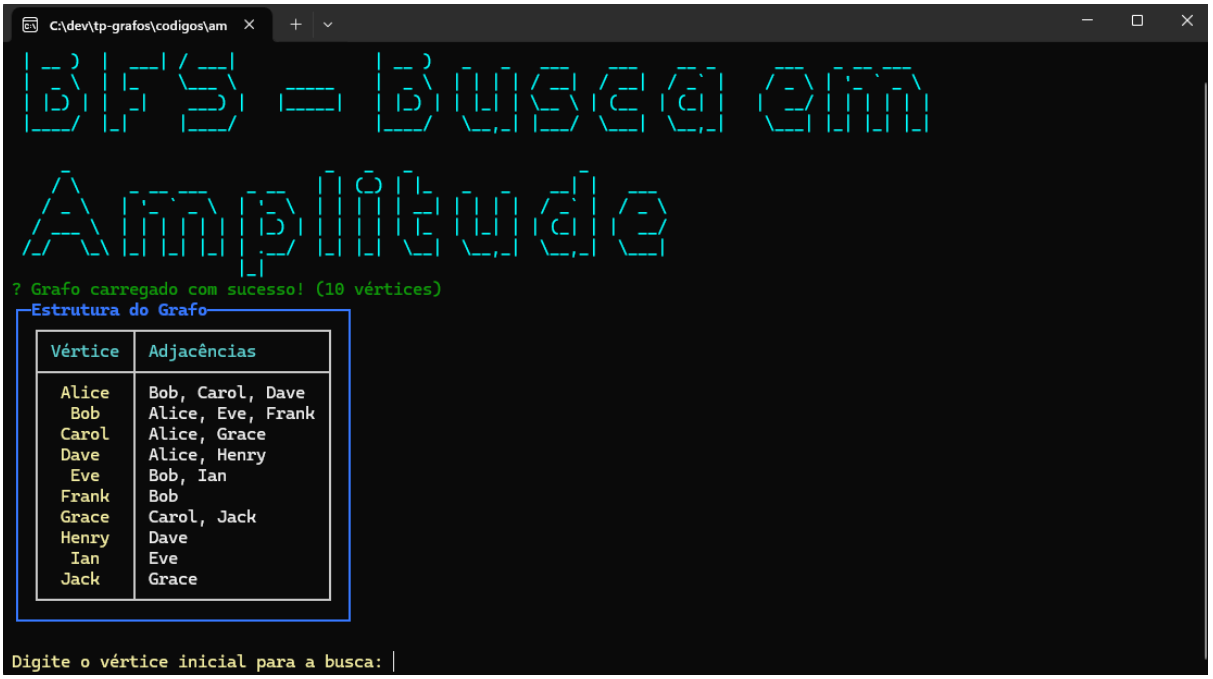
1. Rede Social

Exemplo de conexões:

- Alice conectada a Bob e Carol

- Bob conectado a Eve e Frank

- Carol ligada a Grace



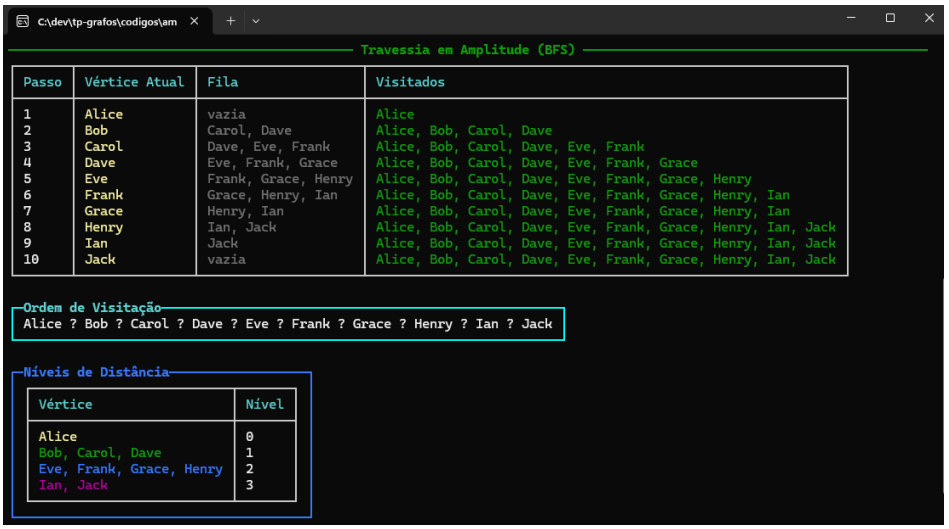
Resultado da BFS a partir de Alice:

- Nível 0: Alice

- Nível 1: Bob, Carol, Dave

- Nível 2: Eve, Frank, Grace, Henry

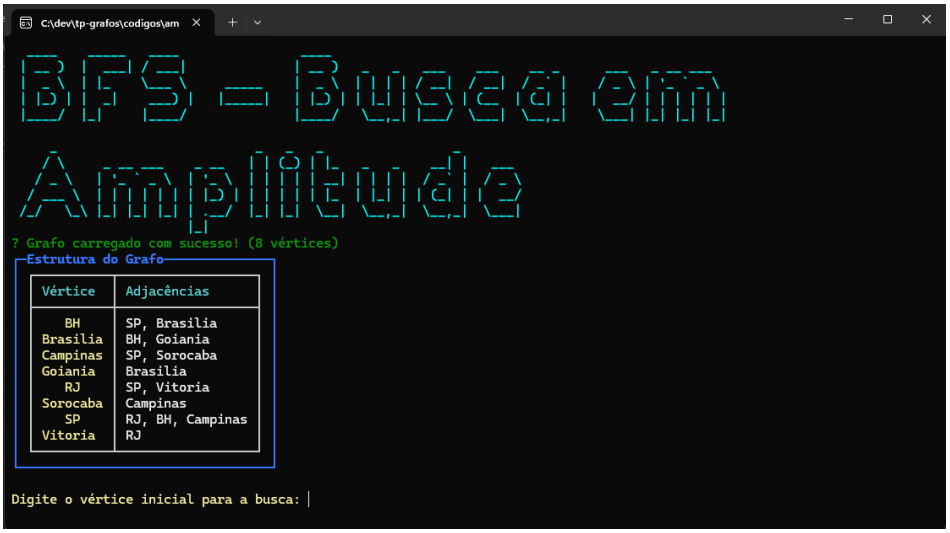
- Nível 3: Ian, Jack



2. Mapa de Cidades

- São Paulo conectada a Belo Horizonte, Campinas e Rio de Janeiro

- Outras cidades conectadas em níveis inferiores



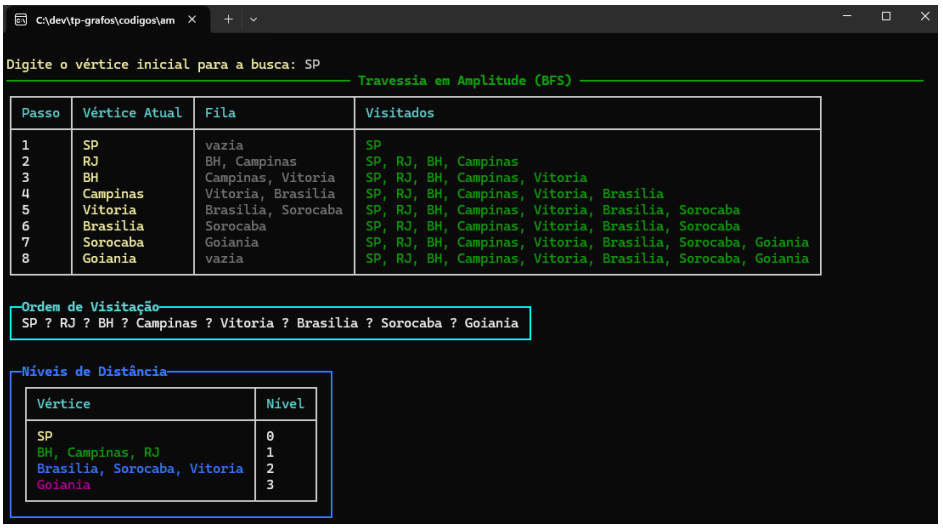
Resultado da BFS iniciando em SP:

- Nível 0: São Paulo

- Nível 1: BH, Campinas, RJ

- Nível 2: Brasília, Sorocaba, Vitória

- Nível 3: Goiânia



Casos de Uso do BFS

- Encontrar menor caminho em grafo não ponderado
- Determinar graus de separação
- Verificar conectividade
- Identificar componentes conexas

Métodos Principais

1. ExecutarBFS() - BFSService.cs

Localização: Services/BFSService.cs
Responsabilidade: Executar o algoritmo de Busca em Amplitude
Resumo: Este é o coração do algoritmo BFS. Ele percorre o grafo partindo de um vértice inicial, visitando todos os vértices alcançáveis nível por nível (Breadth-First).

```
public ResultadoBFS ExecutarBFS(Grafo grafo, string verticeInicial)
    {
        var resultado = new ResultadoBFS();
        var visitados = new HashSet<string>();
        var fila = new Queue<string>();

        fila.Enqueue(verticeInicial);
        visitados.Add(verticeInicial);
        resultado.Niveis[verticeInicial] = 0;

        int numeroPasso = 1;

        while (fila.Count > 0)
        {
            var verticeAtual = fila.Dequeue();
            resultado.OrdemVisitacao.Add(verticeAtual);

            // Registrar o passo atual
            var passo = new PassoBFS
            {
                NumeroPasso = numeroPasso++,
                VerticeAtual = verticeAtual,
                EstadoFila = new List<string>(fila),
                VerticesVisitados = new List<string>(visitados)
            };
            resultado.Passos.Add(passo);

            // Processar adjacentes
            var adjacentes = grafo.ObterAdjacentes(verticeAtual);
            foreach (var adjacente in adjacentes)
            {
```

2. CarregarGrafoDeArquivo() - GrafoService.cs

Localização: Services/GrafoService.cs
Responsabilidade: Ler arquivo e construir estrutura do grafo
Resumo: Lê um arquivo de texto e converte em uma estrutura de dados de grafo (dicionário de adjacências).

```
public Grafo CarregarGrafoDeArquivo(string caminhoArquivo)
    {
        var grafo = new Grafo();

        try
        {
            // Tentar encontrar o arquivo em diferentes localiza��es
            var caminhoCompleto = ResolverCaminhoArquivo(caminhoArquivo);

            if (string.IsNullOrEmpty(caminhoCompleto))
            {
                throw new FileNotFoundException($"Arquivo n�o encontrado: {caminhoArquivo}");
            }

            var linhas = File.ReadAllLines(caminhoCompleto);

            foreach (var linha in linhas)
            {
                if (string.IsNullOrWhiteSpace(linha) || linha.StartsWith("#"))
                    continue;

                var partes = linha.Split(':');
                if (partes.Length != 2)
                    continue;

                var vertice = partes[0].Trim();
                var adjacentes = partes[1].Split(',')
                    .Select(a => a.Trim())
                    .Where(a => !string.IsNullOrEmpty(a))
                    .ToList();

                grafo.Adjacencias[vertice] = adjacentes;
            }
        }
        catch (Exception ex)
        {
            throw new Exception($"Erro ao carregar grafo: {ex.Message}", ex);
        }

        return grafo;
    }
```

3. ContemVertice() - Grafo.cs

Localização: Models/Grafo.cs
Responsabilidade: Verificar se um vértice existe no grafo
Resumo: Método utilizado para validar entrada do usuário antes da execução do BFS.

```
1  public bool ContemVertice(string vertice) => Adjacencias.ContainsKey(vertice);
```

4. ObterAdjacentes() - Grafo.cs

Localização: Models/Grafo.cs
Responsabilidade: Retornar lista de vizinhos de um vértice
Resumo: Método usado pelo BFS para descobrir novos vértices a visitar.

```
1  public List<string> ObterAdjacentes(string vertice)
2   {
3       return Adjacencias.ContainsKey(vertice)
4           ? Adjacencias[vertice]
5           : new List<string>();
6   }
```

5. ExibirResultadoBFS() - BFSUI.cs

Localização: UI/BFSUI.cs
Responsabilidade: Apresentar resultado do BFS no console
Resumo: Transforma o resultado do algoritmo em uma exibição organizada, com tabelas e seções visuais.

Fluxo de Execução Completo:

1. **CarregarGrafoDeArquivo()** → Cria o grafo.
2. **ContemVertice()** → Valida o vértice inicial.
3. **ExecutarBFS()** → Executa o algoritmo.
4. **ObterAdjacentes()** → Consultado a cada iteração.
5. **ExibirResultadoBFS()** → Exibe os resultados.

Conclusão sobre os métodos do BFS:

Os cinco métodos trabalham em conjunto para implementar o algoritmo BFS de forma completa e organizada, respeitando separação de responsabilidades entre camadas de código.

A compreensão desses métodos facilita o entendimento do fluxo do programa e de como o BFS opera na prática.

Algoritmo de Prim

O Algoritmo de Prim é um algoritmo guloso que encontra a Árvore Geradora Mínima (AGM) de um grafo conectado e ponderado. Um bom exemplo é como se você estivesse construindo uma rede de estradas que conecta todas as cidades com o menor custo total possível, adicionando sempre a conexão mais barata disponível. Ele funciona da seguinte maneira:

Começa em um vértice inicial qualquer, mantém um conjunto de vértices já incluídos na árvore, a cada passo seleciona a aresta de menor peso que conecta um vértice da árvore a um vértice fora dela, adiciona essa aresta e o novo vértice à árvore e repete até que todos os vértices estejam incluídos.

Esse algoritmo tem aplicabilidade prática em diversos contextos, sendo eles projeto de redes de telecomunicações, planejamento de redes elétricas, design de circuitos integrados, construção de redes de água ou esgoto, otimização de rotas de cabos submarinos e clustering em problemas de aprendizado de máquina.

Entre suas vantagens, podemos destacar a garantia de encontrar a AGM ótima, funcionamento eficiente com grafos densos quando implementado com heap, complexidade previsível de O(E log V) com heap binário, e facilidade de implementação comparado a outros algoritmos de AGM. Porém ele também possui desvantagens, como funcionar apenas em grafos conectados, exigir que todas as arestas tenham pesos, ser menos eficiente que Kruskal em grafos esparsos e necessitar de estruturas de dados auxiliares (como heap ou fila de prioridade) para boa performance.

- Explicação dos principais trechos de código:

```
public class Program
{
    0 referências
    private static void Main()
    {
        string caminho = "TextFile1.txt";
        int[,] grafo = LerGrafoDoArquivo(caminho);

        var mst = Prim(grafo);

        AnsiConsole.MarkupLine("[bold underline green]Árvore Geradora Mínima (Prim)[/]");
        var tree = new Tree("[yellow]Grafo[/]");

        foreach (var e in mst)
            tree.AddNode($"[cyan]{e.origem}[/] - [cyan]{e.destino}[/]  (peso: [green]{e.peso}[/])");

        AnsiConsole.Write(tree);
    }
```

Esse trecho de código lê um grafo de um arquivo de texto, executa o algoritmo de Prim para encontrar a árvore geradora mínima, e então exibe o resultado no console de forma formatada usando a biblioteca AnsiConsole, mostrando cada aresta da AGM com sua origem, destino e peso em uma estrutura visual de árvore colorida.

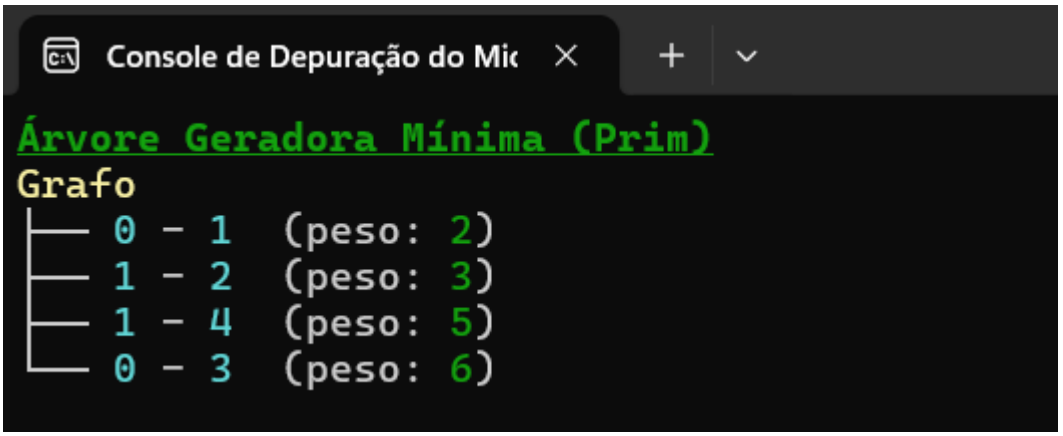
```
private static int[,] LerGrafoDoArquivo(string caminho)
{
    var linhas = File.ReadAllLines(caminho);
    int n = linhas.Length;
    int[,] matriz = new int[n, n];

    for (int i = 0; i < n; i++)
    {
        var valores = linhas[i].Split(' ', StringSplitOptions.RemoveEmptyEntries);

        for (int j = 0; j < n; j++)
            matriz[i, j] = int.Parse(valores[j]);
    }

    return matriz;
}
```

Essa função lê todas as linhas de um arquivo de texto, determina o tamanho da matriz pelo número de linhas, cria uma matriz quadrada de inteiros, e então percorre cada linha dividindo seus valores por espaços e convertendo cada número para inteiro, preenchendo a matriz de adjacências que representa o grafo com seus pesos, retornando essa matriz no final.

```
private static List<(int origem, int destino, int peso)> Prim(int[,] grafo)
{
    int n = grafo.GetLength(0);
    bool[] visitado = new bool[n];
    List<(int origem, int destino, int peso)> resultado = new();

    visitado[0] = true;

    for (int k = 0; k < n - 1; k++)
    {
        int menor = int.MaxValue;
        int origem = -1;
        int destino = -1;

        for (int i = 0; i < n; i++)
        {
            if (!visitado[i]) continue;

            for (int j = 0; j < n; j++)
            {
                if (!visitado[j] && grafo[i, j] != 0 && grafo[i, j] < menor)
                {
                    menor = grafo[i, j];
                    origem = i;
                    destino = j;
                }
            }
        }

        if (destino == -1)
            break; // Grafo desconexo

        visitado[destino] = true;
        resultado.Add((origem, destino, menor));
    }

    return resultado;
```

Essa função implementa o algoritmo de Prim começando pelo vértice zero e o marcando como visitado. A cada iteração, ele percorre todos os vértices já visitados e examina as arestas para vértices não visitados, procurando a aresta de menor peso que conecta a árvore em construção a um vértice externo. Quando encontra essa aresta de peso mínimo, marca o vértice de destino como visitado e adiciona a aresta a lista de resultados. O processo se repete até que todos os vértices estejam incluídos na árvore geradora mínima, ou até detectar que o grafo é desconexo caso não encontre mais arestas válidas, retornando ao final a lista de arestas que formam a AGM.

- Resultados obtidos



A execução do programa mostrou como o algoritmo de Prim construiu a árvore geradora mínima do grafo. Primeiro, o algoritmo começou pelo vértice 0 e selecionou a aresta de menor peso disponível, conectando-se ao vértice 1 com peso 2. Em seguida, expandiu a árvore adicionando o vértice 2 através da aresta 1-2 com peso 3, depois incluiu o vértice 4 pela aresta 1-4 com peso 5, e finalmente conectou o vértice 3 através da aresta 0-3 com peso 6. O resultado final mostrou que a árvore geradora mínima possui peso total de 16, conectando todos os cinco vértices do grafo através de quatro arestas escolhidas de forma gulosa, sempre priorizando a menor conexão disponível entre os vértices já incluídos na árvore e os ainda não visitados, evidenciando a característica fundamental do algoritmo de construir incrementalmente a solução ótima.

Algoritmo de Dijkstra

O Algoritmo de Dijkstra é um algoritmo guloso que encontra o caminho mais curto entre um vértice de origem e todos os outros vértices em um grafo ponderado com pesos não-negativos. Funciona dessa forma:

Começa em um vértice origem com distância zero, inicializa as distâncias de todos os outros vértices como infinito, mantém um conjunto de vértices ainda não processados, a cada passo seleciona o vértice não processado com a menor distância conhecida, atualiza as distâncias dos seus vizinhos se encontrar um caminho mais curto passando por ele e repete até que todos os vértices tenham sido processados.

Esse algoritmo tem aplicabilidade prática em diversos contextos, sendo eles sistemas de navegação GPS e aplicativos de mapas, roteamento de pacotes em redes de computadores, planejamento de rotas de entregas e logística, jogos para cálculo de pathfinding de personagens, sistemas de transporte público para otimizar trajetos e análise de redes sociais para medir proximidade entre usuários.

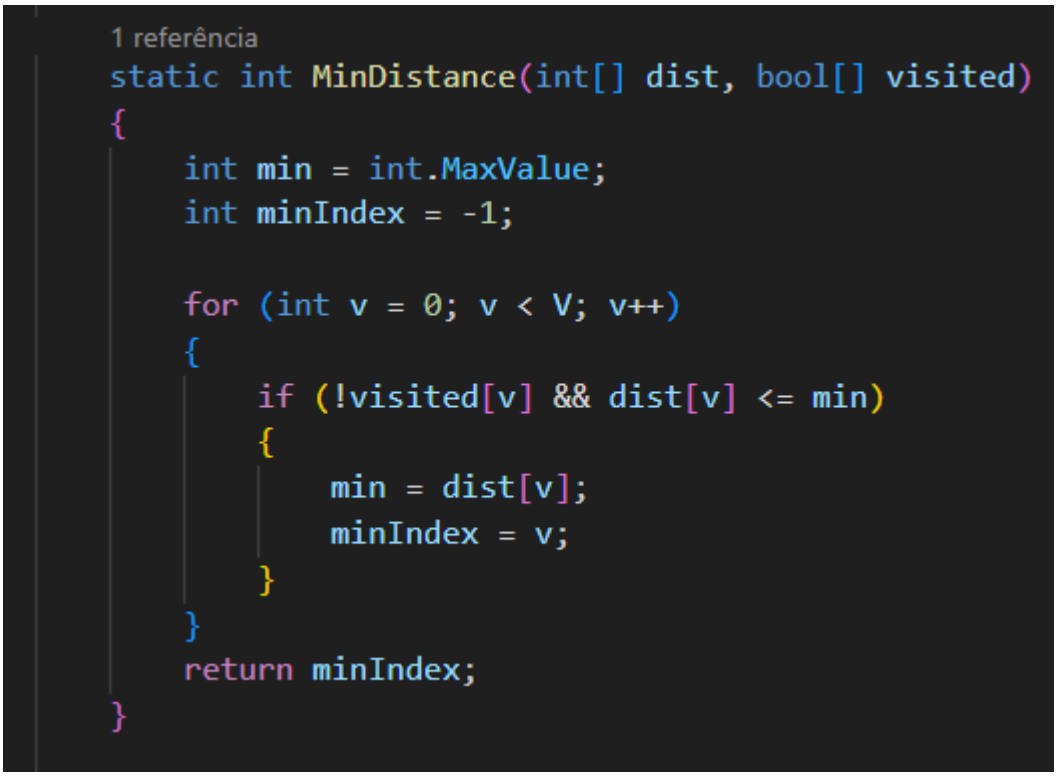
Entre suas vantagens, podemos destacar a garantia de encontrar o caminho mais curto ótimo, funcionamento eficiente quando implementado com fila de prioridade, capacidade de calcular distâncias para todos os vértices em uma única execução, e ampla aplicabilidade em problemas do mundo real.

Porém ele também possui desvantagens, como não funcionar corretamente com arestas de peso negativo, ser menos eficiente que algoritmos especializados quando se busca apenas um destino específico, necessitar de estruturas de dados auxiliares para boa performance e ter complexidade maior que algoritmos mais simples em grafos não ponderados.

```
static void Dijkstra(int[,] graph, int src)
{
    int[] dist = new int[V];
    bool[] visited = new bool[V];

    for (int i = 0; i < V; i++)
    {
        dist[i] = int.MaxValue;
        visited[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++)
    {
        int u = MinDistance(dist, visited);
        visited[u] = true;

        for (int v = 0; v < V; v++)
        {
            if (!visited[v] &&
                graph[u, v] != 0 &&
                dist[u] != int.MaxValue &&
                dist[u] + graph[u, v] < dist[v])
            {
                dist[v] = dist[u] + graph[u, v];
            }
        }
    }
```

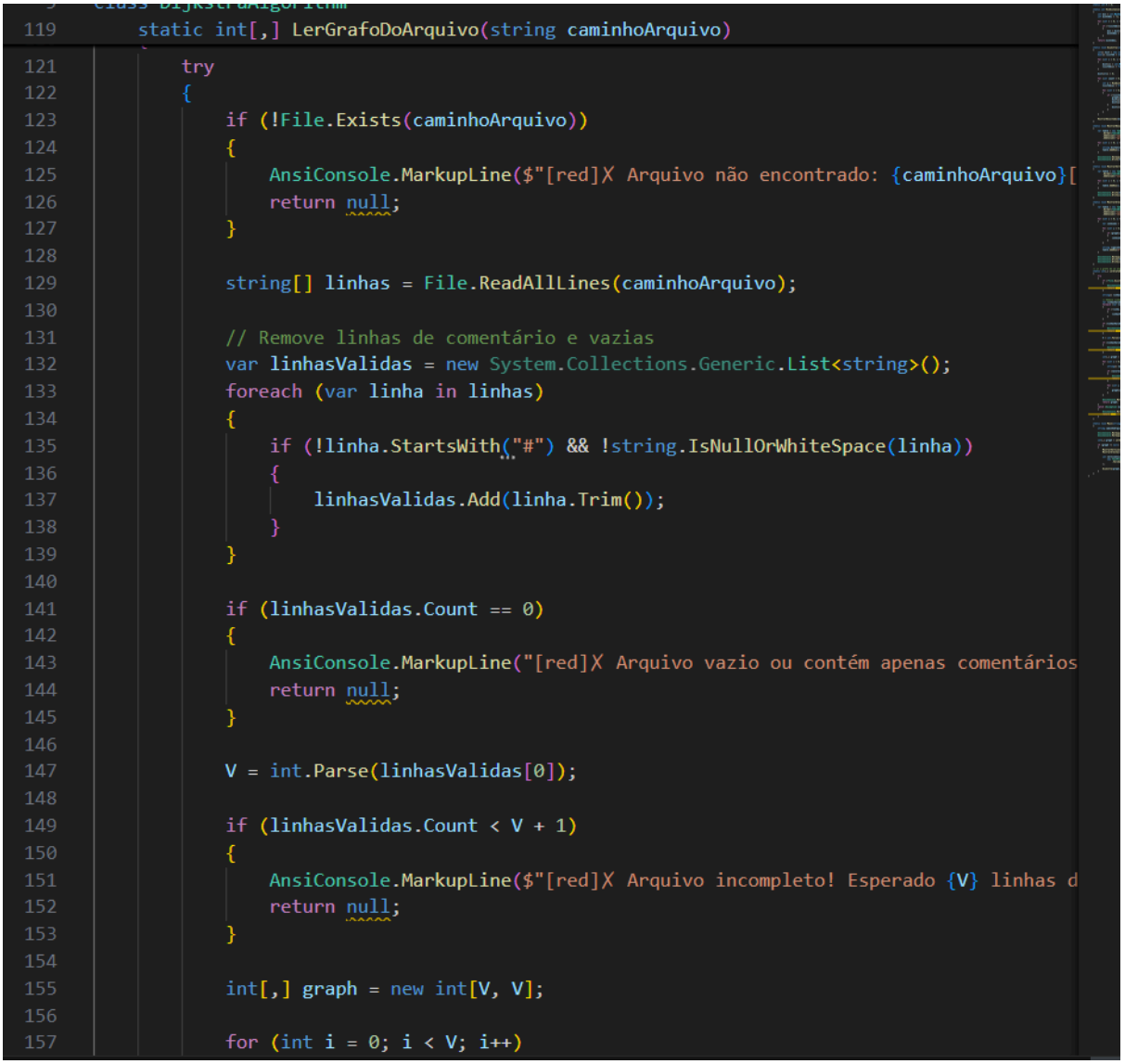
Propósito: Implementa o algoritmo de Dijkstra para calcular a menor distância do vértice src a todos os outros vértices do grafo.
Entrada: graph — matriz de adjacência int[,] com pesos; src — índice do vértice inicial.
Saída: Chama MostrarResultado para exibir as distâncias mínimas (não retorna valor).
Principais passos: Inicializa vetores dist e visited, repete V-1 iterações: escolhe o vértice não visitado com distância mínima (usando MinDistance), relaxa arestas vizinhas atualizando dist.

Observação: Assume arestas não existentes codificadas como 0; não trata pesos negativos.



```
1 referência
static int MinDistance(int[] dist, bool[] visited)
{
    int min = int.MaxValue;
    int minIndex = -1;

    for (int v = 0; v < V; v++)
    {
        if (!visited[v] && dist[v] <= min)
        {
            min = dist[v];
            minIndex = v;
        }
    }
    return minIndex;
}
```

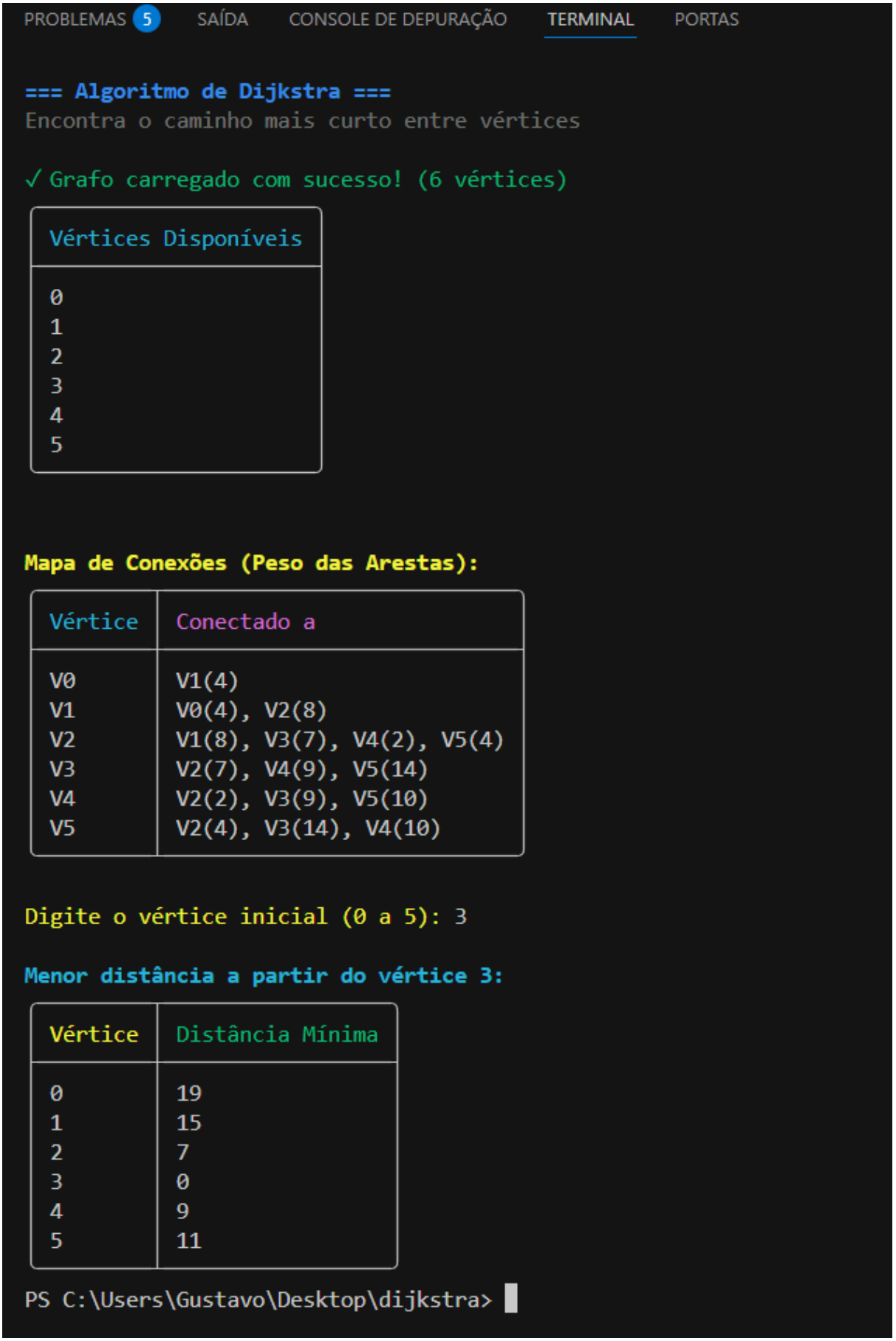
Propósito: Função auxiliar que encontra o índice do vértice não visitado com menor distância atual em distância.
Entrada: dist — array de distâncias; visited — array de flags de visitação.
Saída: Retorna o índice (int) do vértice com menor distância entre os não visitados, ou -1 se não houver.
Principais passos: Percorre todos os vértices, compara dist[v] com o mínimo atual, respeitando visited[v].
Observação: Usa int.MaxValue como infinito; é O(V) por chamada.



```
119     static int[,] LerGrafoDoArquivo(string caminhoArquivo)
121         try
122         {
123             if (!File.Exists(caminhoArquivo))
124             {
125                 AnsiConsole.MarkupLine($"[red]✗ Arquivo não encontrado: {caminhoArquivo}[
126                 return null;
127             }
128
129             string[] linhas = File.ReadAllLines(caminhoArquivo);
130
131             // Remove linhas de comentário e vazias
132             var linhasValidas = new System.Collections.Generic.List<string>();
133             foreach (var linha in linhas)
134             {
135                 if (!linha.StartsWith("#") && !string.IsNullOrWhiteSpace(linha))
136                 {
137                     linhasValidas.Add(linha.Trim());
138                 }
139             }
140
141             if (linhasValidas.Count == 0)
142             {
143                 AnsiConsole.MarkupLine("[red]✗ Arquivo vazio ou contém apenas comentários
144                 return null;
145             }
146
147             V = int.Parse(linhasValidas[0]);
148
149             if (linhasValidas.Count < V + 1)
150             {
151                 AnsiConsole.MarkupLine($"[red]✗ Arquivo incompleto! Esperado {V} linhas d
152                 return null;
153             }
154
155             int[,] graph = new int[V, V];
156
157             for (int i = 0; i < V; i++)
```

static int[,] LerGrafoDoArquivo(string caminhoArquivo):

Propósito: Lê a representação do grafo de um arquivo texto (graph.txt) e constrói a matriz de adjacência.
Entrada: caminhoArquivo — caminho para o arquivo de texto.
Saída: Retorna int[,] com a matriz do grafo, ou null em caso de erro.
Principais passos: Verifica existência do arquivo; remove linhas vazias/comentários (linhas começando com #); interpreta primeira linha como número de vértices V; lê as próximas V linhas como linhas da matriz (valores separados por espaço) e converte para int. Valida formato e exibe mensagens de erro via AnsiConsole.
Observação: Lança mensagens amigáveis para usuário e trata exceções gerais, retornando null em falha.


PROBLEMAS 5
SAÍDA
CONSOLE DE DEPURAÇÃO
TERMINAL
PORTAS
=== Algoritmo de Dijkstra ===
Encontra o caminho mais curto entre vértices
✓ Grafo carregado com sucesso! (6 vértices)
Vértices Disponíveis
0
1
2
3
4
5
Mapa de Conexões (Peso das Arestas):
Vértice
Conectado a
V0
V1(4)
V1
V0(4), V2(8)
V2
V1(8), V3(7), V4(2), V5(4)
V3
V2(7), V4(9), V5(14)
V4
V2(2), V3(9), V5(10)
V5
V2(4), V3(14), V4(10)
Digite o vértice inicial (0 a 5): 3
Menor distância a partir do vértice 3:
Vértice
Distância Mínima
0
19
1
15
2
7
3
0
4
9
5
11
PS C:\Users\Gustavo\Desktop\dijkstra>

Conclusão

O desenvolvimento deste Sistema de Análise de Grafos proporcionou uma compreensão prática dos conceitos fundamentais da teoria dos grafos através da implementação de algoritmos clássicos em C#.

A implementação dos algoritmos DFS e BFS evidenciou as diferentes estratégias de exploração de grafos: enquanto o DFS explora em profundidade, o BFS percorre por níveis. O algoritmo de Prim demonstrou a eficácia de abordagens gulosas para construção de árvores geradoras mínimas, e o algoritmo de Dijkstra consolidou o conceito de relaxamento de arestas para encontrar caminhos mínimos.

O desenvolvimento modularizado, com separação clara entre camadas de serviço, modelo e interface, facilitou a manutenção do código. A utilização da biblioteca Spectre.Console tornou a visualização dos resultados mais clara e profissional.

Os principais desafios enfrentados incluíram a escolha da estrutura de dados adequada (optamos por matriz de adjacência), o controle correto de tempos de descoberta e término no DFS, o tratamento de grafos desconexos no algoritmo de Prim, e a validação robusta na leitura de arquivos de entrada.

Este trabalho demonstrou aplicações práticas da teoria dos grafos em problemas reais e consolidou conceitos teóricos através da implementação. O sistema desenvolvido atende aos objetivos propostos, oferecendo uma plataforma funcional para análise de grafos com código bem estruturado. O trabalho em equipe foi fundamental para o sucesso do projeto, permitindo a divisão de responsabilidades e troca de conhecimentos entre os membros do grupo.

Referências Bibliográficas

Feofiloff, Paulo. *“Algoritmos para Grafos – Algoritmo de Prim”*. IME-USP, 2002 (via Sedgewick: *Algorithms in C, part 5 – Graph Algorithms*). Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/prim.html

Feofiloff, Paulo. *“Algoritmos para Grafos – Busca em profundidade (DFS)”*. IME-USP. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html

Anwar Hermuche. *“Métodos de Busca em Grafos — BFS & DFS”*. Medium, 17 de abril de 2024. Disponível em: https://medium.com/@anwarhermuche/m%C3%A9todos-de-busca-em-grafos-bfs-dfs-cf17761a0dd9

Exponent Labs. *“Breadth-First Search (BFS) – Shortest Paths in Unweighted Graphs”*. Interview Cake. Acesso em 02 de dezembro de 2025. Disponível em: https://www.interviewcake.com/concept/java/bfs

Feofiloff, Paulo. “Algoritmos para Grafos – Busca em largura (BFS)”. IME-USP. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html

W3Schools. “DSA Dijkstra’s Algorithm”. W3Schools. Disponível em: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php