03.12.2024

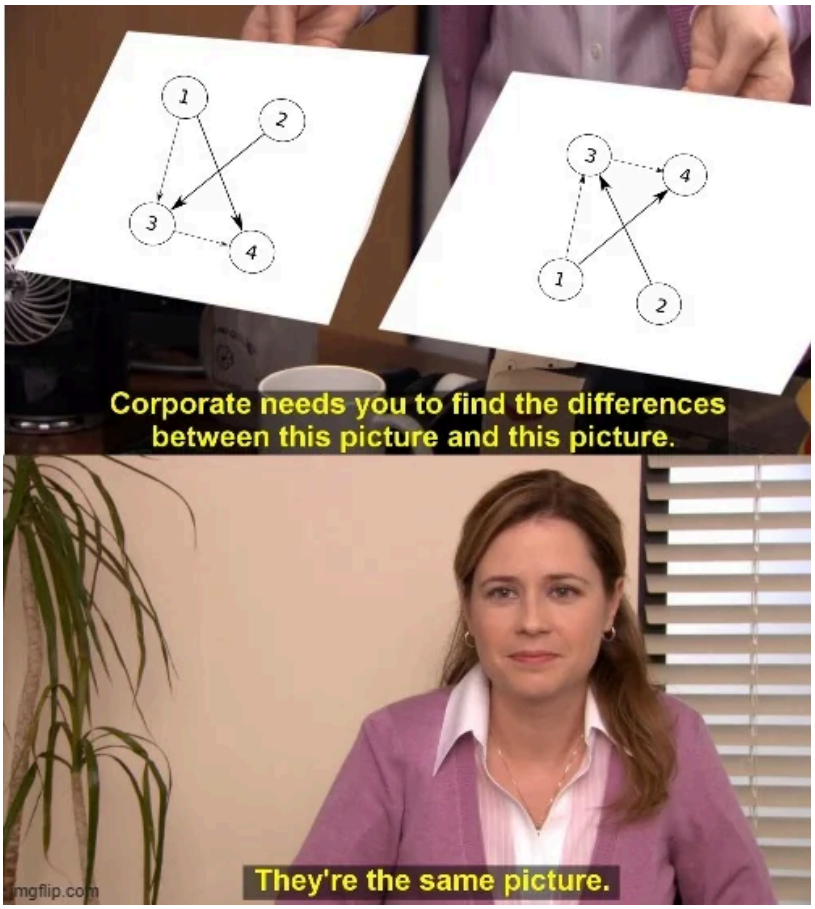
Labs 5-8 - Grafos

Júlia Tadeu - 2312392

Theo Canuto - 2311293

Professor Luiz Fernando Seibel

INF1010 - 3WA



1. Objetivo

O objetivo deste trabalho é implementar e explorar diversos algoritmos e representações de grafos, utilizando o grafo fornecido nas tarefas dos laboratórios 5, 6, 7 e 8. As atividades propostas têm como foco a compreensão e aplicação de conceitos fundamentais de Estruturas de Dados, com ênfase em representações gráficas e algoritmos de busca e otimização.

1. **Representação de Grafos:** Representar o grafo fornecido tanto na forma de lista de adjacências quanto na forma de matriz de adjacências. A representação com lista de adjacências permitirá uma visualização mais eficiente da estrutura do grafo em termos de conexões diretas, enquanto a matriz de adjacências facilitará a implementação de algoritmos que requerem verificações rápidas de existência de arestas.

2. **Busca em Largura (BFS):** Implementar o algoritmo de Busca em Amplitude (BFS) para explorar o grafo, utilizando o vértice 0 como ponto de partida. O objetivo é entender a dinâmica de busca em grafos não ponderados e como as informações de distância e caminho são propagadas através da estrutura do grafo.

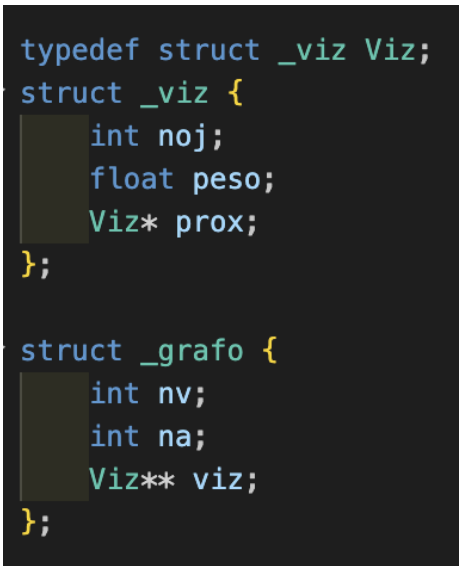
3. **Árvore Geradora Mínima:** Aplicar o algoritmo de Kruskal para gerar a árvore geradora mínima do grafo fornecido, com o intuito de explorar a técnica de otimização que visa a seleção das arestas de menor custo, mantendo a conectividade do grafo.

Essas tarefas visam proporcionar uma compreensão prática sobre as representações de grafos, os algoritmos clássicos de busca e as técnicas de otimização de grafos, especialmente no contexto das árvores geradoras mínimas.

2. Estrutura do Programa

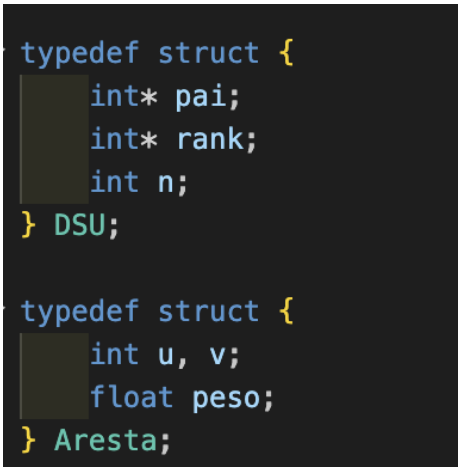
O programa a seguir implementa um grafo ponderado e as operações clássicas em grafos, como a busca em largura (BFS), algoritmo de Kruskal para encontrar a árvore geradora mínima (MST). Além disso, o programa utiliza a estrutura de dados Disjoint Set Union (DSU) para auxiliar no algoritmo de Kruskal e nas operações de união e busca de conjuntos disjuntos. O grafo é representado por listas de adjacência e o código inclui funções para a manipulação dessas listas, bem como para a conversão do grafo para uma matriz de adjacência. O foco do código está na construção e manipulação de grafos com diferentes abordagens de algoritmos para trabalhar com grafos ponderados e suas propriedades, como a MST.

Estruturas de Dados:



```
typedef struct _viz Viz;
struct _viz {
    int noj;
    float peso;
    Viz* prox;
};

struct _grafo {
    int nv;
    int na;
    Viz** viz;
};
```



```
typedef struct {
    int* pai;
    int* rank;
    int n;
} DSU;

typedef struct {
    int u, v;
    float peso;
} Aresta;
```

A estrutura de dados *Viz* representa um vértice na lista de adjacência de um grafo, que contém:

- *noj*: Um número inteiro que representa o índice do nó vizinho.
- *peso*: Um número de ponto flutuante que armazena o peso da aresta que conecta o vértice ao nó vizinho.
- *prox*: Um ponteiro para o próximo vértice na lista de adjacência (se houver mais de um vizinho).

A estrutura de dados *_grafo* representa o grafo com listas de adjacência, que contém:

- *nv*: Um número inteiro que representa o número de vértices do grafo.
- *na*: Um número inteiro que representa o número de arestas no grafo.
- *viz*: Um vetor de ponteiros para a estrutura Viz, que armazena as listas de adjacência de cada vértice.

A estrutura de dados *DSU* representa a estrutura usada para manipulação de conjuntos disjuntos, que contém:

- *pai*: Um vetor de inteiros que armazena o representante (raiz) de cada conjunto.
- *rank*: Um vetor de inteiros que auxilia na união de conjuntos, mantendo a árvore de menor altura.
- *n*: Um número inteiro que armazena o número total de elementos no *DSU*.

A estrutura de dados *Aresta* representa uma aresta no grafo, que contém:

- *u*: Um número inteiro que representa o vértice de origem da aresta.
- *v*: Um número inteiro que representa o vértice de destino da aresta.
- *peso*: Um número de ponto flutuante que representa o peso da aresta.

Função *Viz* novoViz(int noj, float peso)*:

- **Descrição:** Cria um novo vértice na lista de adjacência do grafo, com um nó de destino e o peso da aresta.
- **Parâmetros:** *noj*: Um número inteiro que representa o índice do nó vizinho; *peso*: Um número de ponto flutuante que representa o peso da aresta.
- **Retorno:** Retorna um ponteiro para o novo vértice (*Viz*) criado.

Função *struct _grafo* criaGrafo(int nv):*

- **Descrição:** Cria um grafo com um número de vértices especificado. Inicializa as listas de adjacência para cada vértice.
- **Parâmetros:** *nv*: Um número inteiro que representa o número de vértices no grafo.
- **Retorno:** Retorna um ponteiro para o grafo criado.

Função *void insereAresta(struct _grafo* g, int origem, int destino, float peso):*

- **Descrição:** Insere uma aresta no grafo, atualizando as listas de adjacência de ambos os vértices (origem e destino). Como o grafo é não direcionado, a aresta é inserida em ambas as direções.
- **Parâmetros:** *g:* Ponteiro para a estrutura do grafo onde a aresta será inserida*; origem:* Um número inteiro representando o vértice de origem da aresta; *destino:* Um número inteiro representando o vértice de destino da aresta; *peso:* Um número de ponto flutuante representando o peso da aresta.
- **Retorno:** Não retorna valor.

Função *void imprimeGrafoComEnderecos(struct _grafo* g))*:

- **Descrição:** Imprime o grafo, mostrando os vértices e seus vizinhos, além dos endereços de memória das listas de adjacência.
- **Parâmetros:** *g:* Ponteiro para a estrutura do grafo a ser impresso.
- **Retorno:** Não retorna valor.

Função *void criaMatrizAdjacencias(struct _grafo* g)*:

- **Descrição:** Cria e imprime a matriz de adjacência do grafo, onde o valor da célula [i][j] representa a presença de uma aresta entre os vértices i e j. Caso exista uma aresta, o valor da célula será 1. Caso contrário, o valor será 0.
- **Parâmetros:** *g*: Ponteiro para a estrutura do grafo.
- **Retorno:** Não retorna valor.

Função *void criaListaAdjacencias(struct _grafo* g):*

- **Descrição:** Cria e imprime a lista de adjacência do grafo, mostrando cada vértice e seus vizinhos, junto com os pesos das arestas que os conectam.
- **Parâmetros:** *g:* Ponteiro para a estrutura do grafo.
- **Retorno:** Não retorna valor.

Função *void liberaGrafo(struct _grafo* g):*

- **Descrição:** Libera a memória alocada para o grafo, incluindo as listas de adjacência e o próprio grafo.
- **Parâmetros:** *g:* Ponteiro para a estrutura do grafo a ser liberado.
- **Retorno:** Não retorna valor.

Função *void bfs(struct _grafo* g, int inicio):*

- **Descrição:** Realiza uma busca em largura (BFS) a partir do vértice de início, imprimindo a ordem dos vértices visitados.
- **Parâmetros:** *g:* Ponteiro para a estrutura do grafo; *inicio*: Um número inteiro que representa o vértice de início para a busca.
- **Retorno:** Não retorna valor.

Função *DSU* criaDSU(int n):*

- **Descrição:** Cria uma estrutura DSU com n elementos. Inicializa os pais de cada elemento como ele mesmo e os ranks como 0.
- **Parâmetros:** *n*: Um número inteiro representando o número de elementos no conjunto disjunto.
- **Retorno:** Retorna um ponteiro para a estrutura DSU criada.

Função *int find(DSU* dsu, int x):*

- **Descrição:** Encontra o representante (raiz) do conjunto ao qual o elemento x pertence, utilizando a técnica de compressão de caminho.
- **Parâmetros:** *dsu*: Ponteiro para a estrutura DSU; *x*: Um número inteiro representando o elemento do qual se deseja encontrar o representante.
- **Retorno:** Retorna o índice do representante do conjunto.

Função *void unionDSU(DSU* dsu, int x, int y):*

- **Descrição:** Une os conjuntos de x e y utilizando a técnica de união por rank.
- **Parâmetros:** *dsu*: Ponteiro para a estrutura DSU; *x*: Um número inteiro representando o primeiro elemento; *y*: Um número inteiro representando o segundo elemento.
- **Retorno:** Não retorna valor.

Função *int comparaArestas(const void* a, const void* b):*

- **Descrição:** Função de comparação para ordenar as arestas pelo peso. Usada no algoritmo de Kruskal para ordenar as arestas em ordem crescente de peso.
- **Parâmetros:** a, b: Ponteiros para as duas arestas a serem comparadas.
- **Retorno:** Retorna um valor negativo, zero ou positivo, dependendo do peso das arestas comparadas.

Função *void kruskal(struct _grafo* g):*

- **Descrição:** Implementa o algoritmo de Kruskal para encontrar a árvore geradora mínima (MST) de um grafo. Utiliza a estrutura DSU para controlar a formação de ciclos.
- **Parâmetros:** *g*: Ponteiro para a estrutura do grafo.
- **Retorno:** Não retorna valor.

Função int *main():*

- **Descrição:** Ponto de entrada do programa. Cria o grafo, insere as arestas, imprime o grafo e as suas representações (listas e matrizes de adjacência), realiza a busca em largura (BFS) e aplica o algoritmo de Kruskal para encontrar a árvore geradora mínima (MST). Finalmente, libera a memória alocada.
- **Parâmetros:** Não possui parâmetros.
- **Retorno:** Retorna um inteiro 0, indicando a execução bem-sucedida do programa.

3. Soluções

a. LAB 5:

Matriz de adjacência:

	0	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	0	0	0	0	0	1	0
1	1	0	1	0	0	0	0	1	0
3	0	1	0	1	0	1	0	0	1
4	0	0	1	0	1	1	0	0	0
5	0	0	0	1	0	1	0	0	0
6	0	0	1	1	1	0	1	0	0
7	0	0	0	0	0	1	0	1	1
8	1	1	0	0	0	0	1	0	1
9	0	0	1	0	0	0	1	1	0

Lista de adjacência:

0➡1➡8➡NULL
1➡0➡3➡8➡NULL
3➡1➡4➡6➡9➡NULL
4➡3➡5➡6➡NULL
5➡4➡6➡NULL
6➡3➡4➡5➡7➡NULL
7➡6➡8➡9➡NULL
8➡0➡1➡7➡9➡NULL
9➡3➡7➡8➡NULL

b. LAB 6:

A Busca em Amplitude (BFS) foi implementada utilizando uma abordagem iterativa com a ajuda de uma fila (FIFO). O algoritmo percorre os vértices de um grafo começando a partir de um vértice de origem e explorando todos os seus vizinhos antes de passar para os próximos níveis. A função *bfs* utiliza

uma fila para armazenar os vértices a serem visitados e uma lista booleana para marcar quais vértices já foram visitados. A BFS garante que todos os vértices de uma componente conexa sejam visitados em uma ordem de proximidade ao vértice de origem.

A funcionalidade de verificação de conectividade garante que o grafo esteja conexo, ou seja, que todos os vértices estejam ao menos indiretamente conectados por arestas. O algoritmo utilizado para essa verificação é a Busca em Largura (BFS), que explora o grafo a partir de um vértice inicial e verifica se todos os outros vértices podem ser alcançados.

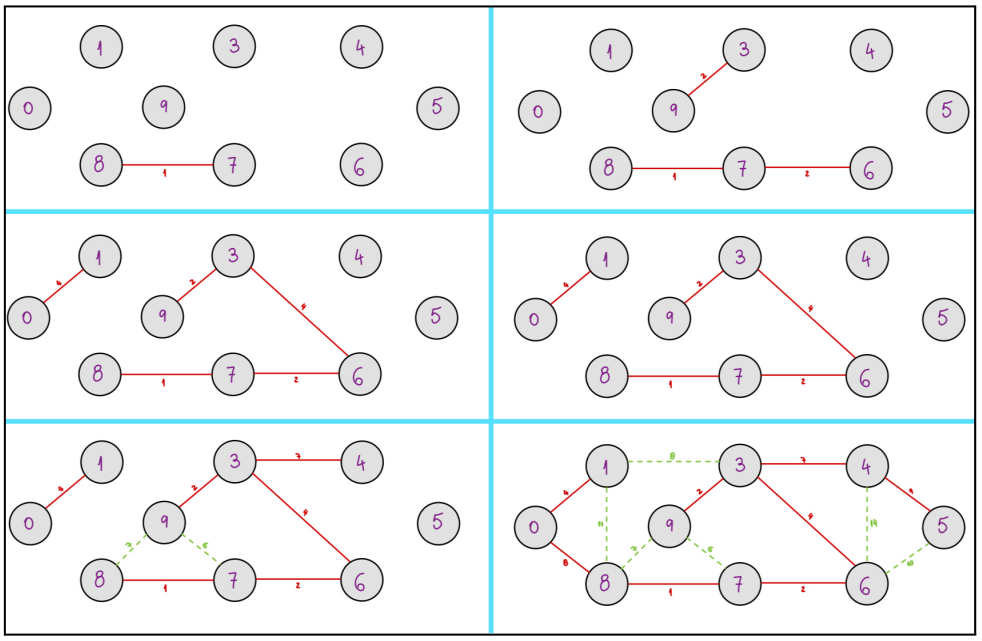
- Exploração de Componentes Conexas: A função bfs permite explorar o grafo a partir de um vértice de início, utilizando uma fila para armazenar os vértices a serem visitados. Durante a execução, o algoritmo marca os vértices como visitados, garantindo que todos os vértices alcançáveis sejam percorridos.

- Verificação de Conectividade: Após a execução da BFS, a verificação de conectividade pode ser feita pela análise do vetor de vértices visitados. Se todos os vértices foram marcados como visitados, o grafo é considerado conexo. Caso contrário, o grafo é desconexo, ou seja, possui componentes conexas independentes.

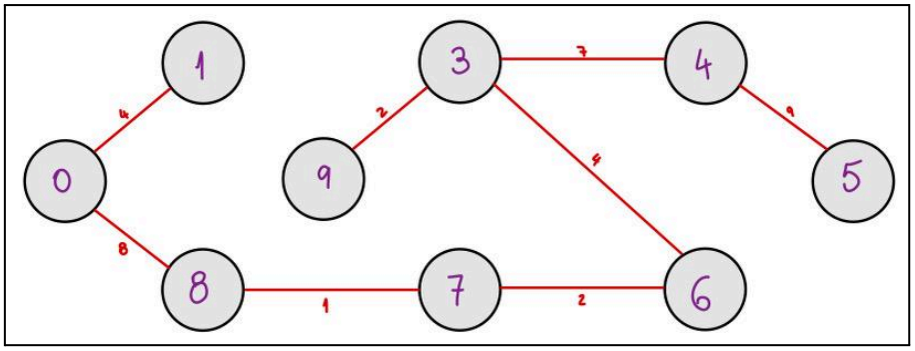
- Identificação de Componentes Conexas: A BFS também pode ser usada para identificar componentes conexas em grafos desconexos. Ao iniciar uma BFS a partir de um vértice não visitado, o algoritmo explora uma componente conexa do grafo, marcando os vértices da mesma. O processo é repetido para os vértices restantes até que todas as componentes conexas sejam encontradas.

c. LAB 7:

O algoritmo de Kruskal foi adotado para encontrar a Árvore Geradora Mínima (MST) do grafo fornecido. Esse algoritmo considera cada nó como uma árvore separada (formando uma floresta), também examina as arestas de menor custo e, nos casos em que estas unem duas árvores na floresta, incluem-nas. Esse processo se repete até que todos os nós estejam conectados. Na imagem abaixo, as linhas vermelhas representam as ligações de fato. Já as tracejadas, situações em que nos deparamos com uma menor ligação que não pode ser feita.



Árvore Geradora Mínima:



d. LAB 8:

- Estrutura de Dados do Grafo
  Para representar o grafo, foi utilizada uma estrutura que armazena o número de vértices, as arestas e seus respectivos pesos. O grafo foi implementado utilizando duas representações: matriz de adjacência e lista de adjacência, permitindo maior flexibilidade nos algoritmos a serem aplicados.

  - Criação do Grafo: A função create_graph inicializa o grafo com um número fixo de vértices e cria uma lista ou matriz de adjacência, conforme a escolha do usuário ou do algoritmo a ser testado.

  - Inserção de Arestas: A função add_edge insere as arestas entre os vértices, incluindo o peso de cada aresta. A função é capaz de adicionar arestas bidirecionais (para grafos não direcionados), conectando ambos os vértices de cada aresta.

- Árvore Geradora Mínima (MST) via Kruskal

O algoritmo de Kruskal foi implementado, utilizando a estratégia de união e find (DSU - Disjoint Set Union) para garantir que nenhuma aresta forme ciclos enquanto o algoritmo constroi a MST.

  - Ordenação das Arestas: A função kruskal_mst começa ordenando todas as arestas do grafo em ordem crescente de peso. Em seguida, ela percorre as arestas e as adiciona à árvore geradora mínima, desde que não formem ciclos, utilizando a estrutura de dados de conjuntos disjuntos (DSU) para verificar a existência de ciclos.

  - Conjuntos Disjuntos (DSU): A estrutura de dados DSU é utilizada para verificar se os vértices das arestas que estão sendo consideradas para inclusão na MST pertencem a componentes conexas distintas. Caso contrário, a aresta é descartada.

- Representação do Grafo

A funcionalidade de impressão do grafo permite visualizar a estrutura do grafo em duas representações: lista de adjacência e matriz de adjacência.

- A matriz de adjacência é uma tabela 2D onde cada posição [i][j] contém o valor 1 se existir uma aresta entre os vértices i e j, e 0 caso contrário. Essa representação é útil para analisar a conectividade entre os vértices e identificar se um vértice está diretamente conectado a outro.
- Lista de Adjacência: A lista de adjacência é uma coleção de listas encadeadas, onde cada lista interna contém os vértices vizinhos de um vértice, juntamente com o peso das arestas que os conectam. Essa representação é mais eficiente em termos de espaço quando o grafo é esparso, ou seja, possui muitas arestas faltando.

Testes e soluções:

Foram realizados testes utilizando grafos com diferentes estruturas e pesos para garantir que os algoritmos implementados funcionassem corretamente. Abaixo estão alguns exemplos de cenários de teste:

- **Cenário 1:** Teste da impressão do grafo com endereços de memória

Entrada: Grafo com os vértices e as arestas inseridas conforme o enunciado.
Esperado: A saída deve exibir o grafo com os vértices e seus respectivos vizinhos, além dos endereços de memória dos destinos das arestas.
Saída:

```
Vertice 0 (Endereco de memoria do vertice: 012DEE80) -> (Destino: 8, Endereco Destino: 012DEEA0)
 (Destino: 1, Endereco Destino: 012DEE84)
Vertice 1 (Endereco de memoria do vertice: 012DEE84) -> (Destino: 3, Endereco Destino: 012DEE8C)
 (Destino: 8, Endereco Destino: 012DEEA0) (Destino: 0, Endereco Destino: 012DEE80)
Vertice 3 (Endereco de memoria do vertice: 012DEE8C) -> (Destino: 9, Endereco Destino: 012DEEA4)
 (Destino: 4, Endereco Destino: 012DEE90) (Destino: 6, Endereco Destino: 012DEE98) (Destino: 1,
Endereco Destino: 012DEE84)
Vertice 4 (Endereco de memoria do vertice: 012DEE90) -> (Destino: 6, Endereco Destino: 012DEE98)
 (Destino: 5, Endereco Destino: 012DEE94) (Destino: 3, Endereco Destino: 012DEE8C)
Vertice 5 (Endereco de memoria do vertice: 012DEE94) -> (Destino: 6, Endereco Destino: 012DEE98)
 (Destino: 4, Endereco Destino: 012DEE90)
Vertice 6 (Endereco de memoria do vertice: 012DEE98) -> (Destino: 7, Endereco Destino: 012DEE9C)
 (Destino: 5, Endereco Destino: 012DEE94) (Destino: 4, Endereco Destino: 012DEE90) (Destino: 3,
Endereco Destino: 012DEE8C)
Vertice 7 (Endereco de memoria do vertice: 012DEE9C) -> (Destino: 9, Endereco Destino: 012DEEA4)
 (Destino: 8, Endereco Destino: 012DEEA0) (Destino: 6, Endereco Destino: 012DEE98)
Vertice 8 (Endereco de memoria do vertice: 012DEEA0) -> (Destino: 7, Endereco Destino: 012DEE9C)
 (Destino: 9, Endereco Destino: 012DEEA4) (Destino: 1, Endereco Destino: 012DEE84) (Destino: 0,
Endereco Destino: 012DEE80)
Vertice 9 (Endereco de memoria do vertice: 012DEEA4) -> (Destino: 7, Endereco Destino: 012DEE9C)
 (Destino: 3, Endereco Destino: 012DEE8C) (Destino: 8, Endereco Destino: 012DEEA0)
```

- **Cenário 2:** Teste da matriz de adjacência e lista de adjacências (LAB 5)

Entrada: Grafo com os vértices e as arestas inseridas conforme o enunciado.
Esperado: A matriz e a lista de adjacência será exibida conforme o esperado.
Saída:

```
Lista de Adjacencias:
Vertice 0: 1 -> 8 -> NULL
Vertice 1: 0 -> 3 -> 8 -> NULL
Vertice 3: 1 -> 4 -> 6 -> 9 -> NULL
Vertice 4: 3 -> 5 -> 6 -> NULL
Vertice 5: 4 -> 6 -> NULL
Vertice 6: 3 -> 4 -> 5 -> 7 -> NULL
Vertice 7: 6 -> 8 -> 9 -> NULL
Vertice 8: 0 -> 1 -> 7 -> 9 -> NULL
Vertice 9: 3 -> 7 -> 8 -> NULL
```

```
Matriz de Adjacencias:
0 1 0 0 0 0 0 1 0
1 0 1 0 0 0 0 1 0
0 1 0 1 0 1 0 0 1
0 0 1 0 1 1 0 0 0
0 0 0 1 0 1 0 0 0
0 0 1 1 1 0 1 0 0
0 0 0 0 0 1 0 1 1
1 1 0 0 0 0 1 0 1
0 0 1 0 0 0 1 1 0
```

- **Cenário 3:** Algoritmo de busca em amplitude (LAB 6)

Entrada: Grafo com os vértices e as arestas inseridas conforme o enunciado.
Esperado: A saída deve exibir a ordem correta BFS.
Saída:

```
BFS a partir do vertice 0:
0 (Endereco: 014BE928)  1 (Endereco: 014BE92C)  8 (Endereco: 014BE948)
3 (Endereco: 014BE934) 7 (Endereco: 014BE944)   9 (Endereco: 014BE94C)
4 (Endereco: 014BE938)  6 (Endereco: 014BE940) 5 (Endereco: 014BE93C)
```

- **Cenário 4:** Algoritmo de Kruskal (LAB 8)

Entrada: Grafo com os vértices e as arestas inseridas conforme o enunciado.
Esperado: Resultado conforme a imagem de esboço do LAB 7.
Saída:

```
Arvore Geradora Minima (MST) - Kruskal:
Aresta (7 [Endereco: 012DEE9C], 8 [Endereco: 012DEEA0]) - Peso: 1.0
Peso Total Atual da MST: 1.0
Aresta (3 [Endereco: 012DEE8C], 9 [Endereco: 012DEEA4]) - Peso: 2.0
Peso Total Atual da MST: 3.0
Aresta (6 [Endereco: 012DEE98], 7 [Endereco: 012DEE9C]) - Peso: 2.0
Peso Total Atual da MST: 5.0
Aresta (3 [Endereco: 012DEE8C], 6 [Endereco: 012DEE98]) - Peso: 4.0
Peso Total Atual da MST: 9.0
Aresta (0 [Endereco: 012DEE80], 1 [Endereco: 012DEE84]) - Peso: 4.0
Peso Total Atual da MST: 13.0
Aresta (3 [Endereco: 012DEE8C], 4 [Endereco: 012DEE90]) - Peso: 7.0
Peso Total Atual da MST: 20.0
Aresta (1 [Endereco: 012DEE84], 3 [Endereco: 012DEE8C]) - Peso: 8.0
Peso Total Atual da MST: 28.0
Aresta (4 [Endereco: 012DEE90], 5 [Endereco: 012DEE94]) - Peso: 9.0
Peso Total Atual da MST: 37.0
Peso Total da MST: 37.0
```

4. Observações e conclusões

Durante a implementação das funcionalidades solicitadas nos laboratórios sobre grafos, incluindo as representações em lista e matriz de adjacências, os algoritmos de Busca em Amplitude e Kruskal, diversos pontos importantes foram identificados:

Facilidades:

Entre as facilidades, destaca-se o uso de estruturas simples para representar os grafos, como a matriz e a lista de adjacências. Essas representações são bem conhecidas e de fácil compreensão, permitindo que o desenvolvimento inicial seja ágil e intuitivo. A matriz de adjacências, por exemplo, é particularmente útil para grafos densos, já que seu tempo de acesso às arestas é constante, o que facilita a implementação e a utilização de algoritmos. A lista de adjacências, além de eficiente para grafos, facilita a implementação de algoritmos como a Busca em Largura (BFS), uma vez que a exploração das arestas se torna mais direta e otimizada em termos de uso de memória.

Dificuldade:

O maior ponto de dificuldade foi o tratamento de ciclos e conectividade em grafos. No caso do algoritmo de Kruskal, garantir que as arestas adicionadas não formem ciclos exigiu o uso de estruturas como o conjunto disjunto, que, embora eficiente, exige um bom entendimento do conceito de união e achamento (DSU). Da mesma forma, verificar a conectividade do grafo antes de aplicar os algoritmos de Kruskal pode ser desafiador, uma vez que é necessário realizar uma busca (como BFS) para garantir que o grafo seja conexo antes de proceder com a construção da árvore geradora mínima.

Conclusão:

A implementação das atividades propostas nos laboratórios foi bem-sucedida, permitindo a exploração de diferentes representações de grafos e a aplicação de algoritmos clássicos de busca e otimização. A construção da lista de adjacências e da matriz de adjacências possibilitou uma melhor compreensão das estruturas de dados e suas implicações no desempenho dos algoritmos. O algoritmo de Busca em Amplitude (BFS) foi implementado corretamente, permitindo a exploração do grafo a partir do vértice 0, conforme o esperado. Além disso, a implementação do algoritmo de Kruskal para a geração da árvore geradora mínima demonstrou a eficácia das abordagens de otimização para grafos ponderados.

O programa se comportou adequadamente nos testes realizados, atingindo os objetivos de análise e otimização de grafos propostos nos laboratórios. No geral, o trabalho permitiu uma compreensão aprofundada dos conceitos de grafos, suas representações e os algoritmos de busca e otimização, contribuindo significativamente para o nosso aprendizado sobre a teoria e a prática de Estruturas de Dados.