Universidade de Brasília

Departamento de Ciência de Computação

Projeto 2: Estudo sobre Grafos

Alunos:

Lucas Hélio Matrícula: 17/0039692

Juliano Balçante Matrícula: 17/0037886

BRASÍLIA – DF

2018

Sumário

1. INTRODUÇÃO

1.1. Objetivo

Este projeto tem por objetivo desenvolver um programa que dado um grafo qualquer, não direcionado e com pesos positivos, calcular o menor caminho entre dois vértices e calcular uma árvore geradora mínima (*Minimum Spanning Tree* - MST) do grafo. Podendo ser utilizado na resolução de problemas que envolvem grafos, como trajeto mínimo entre dois pontos num mapa ou menos a minimização de custo de um problema, como uma rede de distribuição elétrica, por exemplo.

1.2. Delimitação do problema

O programa consiste em, a partir de um grafo de entrada, gerar uma árvore geradora mínima e o menor caminho entre dois vértices.

Caso apenas dois parâmetros sejam passados por linha de comando, ou seja, apenas o nome do programa e um arquivo de grafo, o programa deve seguir os seguintes passos:

1. Calcular o menor caminho entre o vértice 1 e os vértices 10, 100, 1000 e 10000 quando existirem;
2. Calcular a MST do grafo do grafo com vértice 1 como raiz.

Já, em caso de existir um terceiro parâmetro por linha de comando e este for o caractere “1”, então um menu deve ser habilitado para oferecer ao usuário o controle das operações realizadas sobre um grafo. Dessa forma, é ofertada a funcionalidade de calcular a MST do grafo de entrada a partir de uma raiz qualquer e calcular o menor caminho entre vértices quaisquer.

O arquivo que descreve um grafo deve conter as informações do tamanho do grafo (quantidade de vértices) e a descrição de cada aresta existente: vértices unidos pela aresta e seu seguinte peso. Não são admitidos pesos negativos para arestas, assim como índice de vértices negativos ou maiores que o tamanho do grafo. Por padrão, este programa não executa grafos com ciclos, sendo estas arestas desconsideradas na etapa de inclusão de arestas. Além disso, caso o grafo seja desconexo, pode não haver solução para o menor caminho entre dois vértices e a árvore geradora mínima não alcança todos os vértices (consequência do Algoritmo de Prim).

O menor caminho entre dois vértices é calculado por meio do Algortimo de Dijkstra, que não assume pesos negativos. Já a MST é calculada pelo algoritmo de Prim, que pode não alcançar todos os vértices a partir de um vértice raiz (em caso de um grafo desconexo).

A saída padrão do programa consiste em exibir o caminho mínimo entre o vértice 1 e os vértices 10, 100, 1000 e 10000 (se existirem), informando, para cada caminho, a quantidade de vértices que compõe o caminho, cada aresta e seu respectivo peso e, por fim, o peso total. Em seguida deve exibir a MST gerada informando a quantidade de vértices que a compõe, cada aresta e seu respectivo peso e, em seguida, o peso total da MST.

1.3. Metodologia de desenvolvimento

O programa funciona a partir de um grafo contido em um arquivo. O programa foi modularizado usando tipos abstratos de dados (TADs), pois permite que se realize operações sobre as TADs de forma isolada, apenas passando como parâmetro as variáveis corretas e conhecendo o tipo de variável que uma dada função retorna. Além disso, optou-se pela representação de um grafo utilizando lista de adjacências.

O programa deve ser compilado por meio do arquivo Makefile. Todas as estruturas de dados são alocadas dinamicamente e desalocadas ao final do programa.

1.4. Estrutura do projeto

1.4.1. Descrição dos módulos e interdependência

O programa foi dividido em 5 módulos: módulo de grafo, módulo dijkstra, módulo mst, módulo de entrada de dados e módulo de saída de dados.

Os principais módulos do programa são o módulo de grafo, o módulo dijkstra e o módulo mst, já que o primeiro define uma estrutura de grafo e os dois últimos executam operações sobre um grafo. Os módulos restantes, que tratam o fluxo de dados do programa são o módulo de entrada e saída de dados e o módulo de impressão, que são módulos secundários com a função apenas de gerenciar o fluxo de dados.

O módulo de grafo foi desenvolvido no arquivo grafo.c e sua interface está em grafo.h. A função desse módulo é realizar operações sobre a TAD Grafo, definida em um struct Grafo que contém um tamanho, uma quantidade de arestas e uma lista de vértices do tipo Vertice, que por sua vez é definido numa struct Vertice que contém informações do grau, anterior, valor, statusMST e uma lista de adjacência. A lista de adjacência é definida numa struct Adjacencia que contém um peso e um destino. As operações que o módulo pode executar são: criar um grafo, criar arestas, verificar condições de validade de arestas, adicionar arestas, alocar ou realocar espaço para inserção de arestas, realizar buscas por vértices numa lista de adjacência, atribuir tamanho, quantidade de arestas ou incrementar o grau de um vértice e obter o estado de

um grafo, de arestas e de vértices, como tamanho, quantidade de arestas, peso de uma aresta e grau de um vértice. Como este é o módulo base, não depende de nenhum outro.

O módulo dijkstra foi desenvolvido no arquivo dijkstra.c e sua interface está em dijkstra.h. Neste módulo se encontram as funções necessárias ao funcionamento do algoritmo de Dijkstra, como uma função para aplicar relaxação, verificar se ainda existem vértices não visitados, inicializar os vértices como não visitados e com distância infinita.

O módulo mst foi implementado no arquivo mst.c e sua interface está em mst.h. Este módulo define uma struct Aresta com três variáveis: vértice origem, vértice destino e peso, para guardar as arestas selecionadas pelo algoritmo de Prim. Além disso, contém as funções para executar o algoritmo de Prim e encontrar o vértice com menor valor (durante a execução do algoritmo de Prim). Este módulo depende do módulo de grafo.

O módulo de impressão foi implementado no arquivo pritter.c e sua interface está em printter.h. Este módulo contém as funções que permitem imprimir um grafo (com todas as arestas que partem de todos os vértices), uma única aresta, a lista de distâncias geradas pelo algoritmo de Dijkstra, imprimir o caminho entre um vértice e outro e imprimir a árvore geradora mínima encontrada pelo algoritmo de Prim. Este módulo depende do módulo de grafo, do módulo Dijkstra, do módulo mst.

O módulo de entrada e saída foi implementado no arquivo data.c e sua interface está em data.h. Este módulo contém uma função para ler os dados de um grafo ao realizar uma leitura de um arquivo com endereço passado por linha de comando, ao passo que se adiciona as arestas ao grafo durante a leitura. Também tem duas funções de saída de dados que fornecem uma saída no padrão conforme os parâmetros passados por linha de comando; isso será melhor esclarecido na secção 2.4. Este módulo depende do módulo de grafo, do módulo mst e do módulo Dijkstra e do módulo de impressão.

1.4.2. Tipos abstratos de dados

Os módulos foram desenvolvidos nos seguintes arquivos: data.c, data.h, grafo.c, grafo.h, dijkstra.c, dijkstra.h, mst.c, mst.h, printter.c, printter.h. Nos arquivos com extensão .h estão a definição das estruturas utilizadas e a interface de acesso às operações permitidas e nos arquivos de extensão .c estão as próprias operações implementadas.

No módulo de grafo tem-se as seguintes operações:

criaGrafo: cria um grafo vazio;

adicionaAresta: cria uma aresta entre dois vértices

add: realiza operações de alocação dinâmica para alocar espaço para as arestas que são inseridas a medida que o grafo vai sendo construído.

addAlocando: aloca espaço para adicionar uma aresta a um vértice (que deve ter grau nulo).

addRealocando: realoca espaço para adiciona uma aresta a um vértice que já possui uma aresta, ou seja, possui grau diferente de zero.

validaGrafo: verifica condições que invalidam uma aresta para os grafos tratados pelo programa, como pesos negativos ou arestas que são ciclos.

buscaVerticeAdj: realiza uma busca por um vértice v na lista de adjacência de um vértice u, retornando o índice do vértice procurado na tal lista. Retorna -1 como insucesso.

getSize: retorna o tamanho do grafo

getEdge: retorna a quantidade de arestas de um grafo

getGrau: retorna o grau de um vértice

getPeso: retorno o peso de uma aresta

setEdge: atribui uma quantidade de arestas ao grafo

setGrau: atribui um grau a um vértice

nextGrau: incrementa o grau de um vértice

No módulo dijkstra tem-se as seguintes funções:

dijkstra: executa o algoritmo de Dijkstra retornando uma lista de distâncias entre um vértice e todos os outros.

relaxa: aplica uma relaxação se necessário

menorDist: retorna o índice do vértice com menor distância na lista de distâncias

existeAberto: verifica se ainda existe vértice não visitado (aberto)

inicializaD: inicializa as distâncias como infinito

inicializaAberto: inicializa todos os vértices como aberto

Módulo mst:

mstPrim: função que executa o algoritmo de Prim para calcular a MST de um grafo a partir de um vértice raiz. Retorna uma lista de arestas encontradas pelo algoritmo de Prim para compor uma MST.

minimum: retorna o índice do vértice com menor chave (valor)

Módulo de entrada e saída de dados:

recebedados: recebe os dados dos arquivos passados como parâmetro;

saidaPadrao: mostra a saída conforme o padrão definido: MST do grafo de entrada, seguido do menor caminho entre os vértices 1 e 10, 100, 1000 e 10000 quando existirem.

abreMenu: exibe um menu que permite ao usuário realizar operações sobre o grafo (calcular MST ou calcular o menor caminho entre dois vértices). Só é habilitada quando se passa o caractere "1" como terceiro parâmetro por linha de comando.

Módulo de impressão:

imprimeGrafo: imprime todas as arestas que partem de cada vértice. Como o grafo é não direcionado, uma mesma aresta é impressa duas vezes.

imprimeAresta: imprime o conteúdo de uma aresta, isto é, os vértices de origem e destino e o peso.

imprimeDistancia: imprime a lista de distâncias de um vértice a todos os outros.

resultadoDijkstra: imprime o caminho entre um vértice e outro a partir da lista de distâncias calculada pelo algoritmo de Dijkstra.

imprimeMST: imprime todas as arestas encontradas pelo algoritmo de Prim com pesos maiores que zero.

2. DESENVOLVIMENTO DO PROGRAMA

2.1. Solução do problema

O problema consiste em calcular um caminho mínimo entre dois vértices qualquer. Também, dado um vértice raiz, gerar uma MST para o grafo. Para o cálculo do caminho mínimo deve-se usar o algoritmo Dijkstra, já para a MST deve-se utilizar o algoritmo de Prim.

2.1.1. Algoritmo de Dijkstra

Segue abaixo o pseudocódigo do algoritmo de Dijkstra para facilitar o entendimento de como o programa consegue solucionar o problema de encontrar o menor caminho entre um vértice e outro.

```
Dijkstra(Grafo, s){
    para(cada v pertencente a V){
        d[v] := infinito;
    }
    d[s] := 0;
    Q := V;
    enquanto(Q != vazio){
        u = extraiMinimo(Q);
        para(cada v pertencente a u->Adj[]){
            se(d[v] > d[u] + w(u,v)){
                d[v] := d[u] + w(u,v);
            }
        }
    }
}
```

Note que o pseudocódigo acima encontra a distância entre um vértice a todos os outros.

2.1.2. **Algoritmo de Prim**

O pseudocódigo abaixo descreve o funcionamento do algoritmo de Prim para calcular uma árfore geradora mínima de um grafo a partir de um vértice raiz.

```
mstPrim(G, r){
    Q := V;
    para(cada u pertencente a Q){
        chave[u] := infinito;
    }
    chave[r] = 0;
    enquanto(Q != vazio){
        u = extraiMinimo(Q);
        para(cada v pertencente u->adj[]){
            se(v pertence a Q E chave[v] > chave[u] + w(u,v)){
                chave[v] := w(u,v);
            }
        }
    }
}
```

2.2. Atores envolvidos no sistema

O sistema interage com um único ator: usuário. O usuário é quem quer que use o programa para realizar cálculos sobre grafos. O usuário recebe dados tempo de execução de cada chamada

de função para calcular caminho mínimo e gerar MST, além de receber o caminho na forma de uma lista de arestas em sequência e receber a árvore na forma de uma lista de todas as arestas que compõem a MST.

2.3. Entrada do programa

A entrada da linha de comando do programa deve obedecer à seguinte forma:

./nome_programa ArqGrafo <exibição de menu>

Onde:

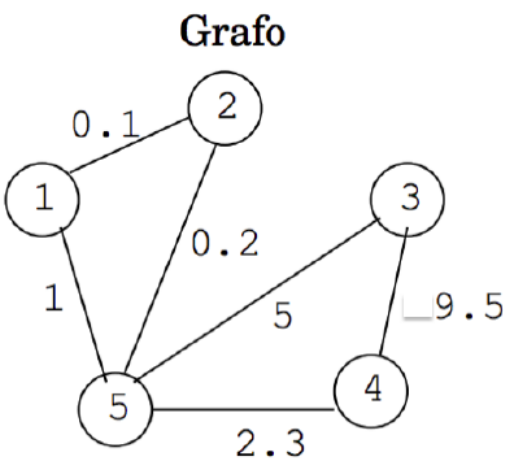
nome_programa é o nome do programa gerado na compilação. Por padrão, “projGrafo”.

ArqGrafo é endereço do arquivo que contém a definição do grafo.

< exibição de menu > é o parâmetro que habilita um menu de operações para se executar sobre o grafo. Para habilitar, atribua a esse parâmetro o caractere “1”. Do contrário, deixe vazio para a execução padrão do programa: calcular menor caminho entre o vértice 1 e os vértices 10, 100, 1000 e 10000 e calcular a MST a partir de um vértice raiz. Ressalta-se que um terceiro parâmetro diferente de “1” não é válido e, por isso, o programa é encerrado.

As entradas dentro do arquivo do grafo devem obedecer ao seguinte padrão: uma linha contendo um inteiro V que define o tamanho do grafo, seguida de E linhas contendo três valores representando arestas onde a e-ésima linha corresponde a e-ésima aresta inserida no grafo. Uma linha contendo os valores de uma aresta deve seguir o seguinte padrão: A B W, onde A e B são inteiros que correspondem aos índices dos vértices unidos pela aresta em questão e W, um número real, corresponde ao peso dessa aresta, tal que $1 \leq A,B \leq V$, $A \neq B$ e $W > 0$.

Por exemplo, o grafo abaixo deve ser descrito pelo seguinte arquivo:



Arquivo de entrada

```
5
1  2  0.1
2  5  0.2
5  3  5
3  4  9.5
4  5  2.3
1  5  1
```

2.4. Saída do programa

A saída do programa no seguinte padrão:

Para cada chamada da função dijkstra(O,D) deve-se imprimir uma saída no seguinte formado:

Uma linha contendo um inteiro E que corresponde a quantidade de arestas existentes no menor caminho entre o vértice O, de origem, ao vértice D, de destino. Seguido de E linhas correspondente as arestas que descrevem o caminho entre os vértices O e D, onde a e-ésia linha contém dois inteiros e uma variável de ponto flutuante no formato A B W, onde A e B corresponde aos vértices unidos pela aresta e W ao peso da aresta. E por último uma linha contendo um número real P correspondente ao peso total do caminho.

Por padrão, a função dijkstra() deve ser chamada para calcular o menor caminho entre os vértices 1 e 10, 100, 1000 e 10000 caso existam, a exceção de quando o usuário habilita um menu de operações sobre grafos, ao passar um segundo parâmetro por linha de comando: “1”, conforme definido na secção 2.3. Em tal caso, a função dijkstra é chamada livremente pelo usuário e para cada chamada há uma saída no formato acima.

Já para cada chamada da função mstPrim(R) deve-se imprimir uma saída no seguinte formato:

Uma linha contendo um inteiro V que corresponde a quantidade de vértices que compõem a árvore (por definição, será a mesma quantidade de vértices do grafo original, mesmo que existam vértices não alcançáveis pela MST), seguida de E linhas que descrevem as arestas da árvore geradora mínima onde a e-ésia linha contém dois inteiros e uma variável de ponto flutuante no formato A B W, onde A e B corresponde aos vértices unidos pela aresta e W ao peso da aresta. E por último uma linha contendo um número real P correspondente ao peso total da árvore.

Por padrão, a função mstPrim() é chamada uma única vez utilizando como vértice raiz 1 . A exceção de quando o usuário solicita um menu, conforme definido na secção 2.3, e pode chamar a função mstPrim() livremente com vários vértices raiz diferentes.

A saída é registrada no diretório “output” nos arquivos “mst.txt” e “dijkstra.txt”. Assim, a cada execução do algoritmo de Dijkstra e de Prim estes arquivos podem ser sobrescritos.

3. EXECUÇÃO DE TESTES

3.1. Metodologia de testes

Os testes foram realizados em duas etapas. A primeira consiste em 5 testes com pequenos grafos de forma a facilitar uma correção manual do resultado do programa, estes são chamados grafos selecionados e estão desenhados no apêndice, no final desse docuemento. Os manuscritos que serviram de correção ao teste estão disponíveis no apêndice. Já segunda etapa corresponde à análise dos grafos disponíveis no moodle da discipliana.

Após verificar e corrigir manualmente os resultados do programa para grafos pequenas (grafos selecionados), pode-se dá prosseguindo às etapas de teste seguintes.

Os arquivos de grafos utilizados se encontram do diretório “data”, onde os grafos selecionados se encontram no diretório “data/GrafosSelecionados”, os grafos do moodle se encontram no diretório “data/GrafosMoodle” e os grafos gerados aleatoriamente se encontram no diretório “data/GrafosAleatorios”.

3.2. Testes realizados com grafos selecionados

3.2.1. Grafo 1

O grafo é descrito pelo arquivo abaixo.

```
9
1 2 0.1
1 5 1
2 4 3
2 3 4
2 5 0.5
3 5 0.9
3 6 8
5 8 0.1
6 9 2
6 7 1
8 9 5
```

Figura 1: Arquivo de entrada

É possível verificar que este é um grafo conexo e a mst pode ser calculada corretamente a partir de qualquer vértice raiz. A mst calculada a partir do vértice 1 é definida no arquivo de saída abaixo, note que o peso da mst é 12,60. A primeira linha indica o tamanho do grafo original, logo indica a quantidade máxima de vértices da mst. Pela saída padrão, como o grafo tem tamanho 9, não existem vértices maiores ou iguais a 10, logo a função Dijkstra não é

chamada nenhuma vez. Entretanto pode-se calcular o menor caminho entre os vértices 1 e 6 e entre os vértices 5 e 7 ao chamar o menu de operações, na passagem de parâmetros para o programa.

```
9
2 1 0.10
5 2 0.50
5 3 0.90
2 4 3.00
9 6 2.00
6 7 1.00
5 8 0.10
8 9 5.00
12.60
```

Figura 2: MST do grafo selecionado 1

```
4
1 2 0.10
2 5 0.50
5 8 0.10
8 9 5.00
9 6 2.00
7.70
```

Figura 3: Menor caminho entre os vértices 1 e 6

```
3
5 8 0.10
8 9 5.00
9 6 2.00
6 7 1.00
8.10
```

Figura 4: Menor caminho entre os vértices 5 e

3.2.2. Grafo 2

A partir do grafo anterior pode-se apagar as arestas entre os vértices 3 e 6 e ente os vértices 5 e 8, de forma a se ter um grafo desconexo. Pode-se analisar nesse teste os efeitos de se utilizar o algoritmo de Prim em grafos desconexos.

```
9
1 2 0.1
1 5 1
2 4 3
2 3 4
2 5 0.5
3 5 0.9
6 9 2
6 7 1
8 9 5
```

Figura 5: Arquivo de entrada

Note que se a MST calculada pelo vértice raiz 1 não alcança os vértices da outra parte desconectada do grafo, assim como a MST calculada a partir do vértice raiz 9.

```
9
1 2 0.10
5 3 0.90
2 4 3.00
2 5 0.50
4.50
```

Figura 6: MST gerada a partir do vértice raiz 1

```
9
9 6 2.00
6 7 1.00
9 8 5.00
8.00
```

Figura 7: MST gerada a partir do vértice raiz 9

3.2.3. Grafo 3

Este grafo testado é completamente conexo e se pode calcular a MST e os menores caminhos tranquilamente. O grafo é descrito no arquivo abaixo.

```
10
1 2 8
2 5 4
2 3 10
3 4 5
4 5 7
5 6 2
4 7 11
6 7 12
6 8 14
8 7 20
6 10 9
6 9 6
10 9 3
9 8 17
```

Figura 8: Arquivo de entrada

Temos a seguinte MST gerada e o menor caminho entre o menor caminho entre 1 e 10.

```
10
1 2 8.00
4 3 5.00
5 4 7.00
2 5 4.00
5 6 2.00
4 7 11.00
6 8 14.00
6 9 6.00
9 10 3.00
60.00
```

Figura 9: MST gerada a partir do vértice 1

```
3
1 2 8.00
2 5 4.00
5 6 2.00
6 10 9.00
23.00
```

Figura 10: Menor caminho entre os vértices 1 e 10

3.2.4. Grafo 4

O grafo está descrito no arquivo abaixo.

```
8
1 2 1
1 3 13
1 4 6
1 6 5
6 8 16
7 8 10
5 8 8
3 8 15
4 2 14
4 7 9
5 2 3
5 7 11
6 4 7
6 7 2
3 2 5
3 5 4
```

Figura 11: Arquivo de entrada

Pode-se encontrar a seguinte MST para o grafo acima.

```
8
1 2 1.00
5 3 4.00
1 4 6.00
2 5 3.00
1 6 5.00
6 7 2.00
5 8 8.00
29.00
```

Figura 12: MST gerada a partir do vértice 1

3.2.5. Grafo 5

E, por fim, esse último teste vem para mostrar o que acontece quando se tentar encontrar a MST a partir de um vértice isolado.

Veja o seguinte grafo descrito no arquivo abaixo.

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

Figura 13: Arquivo de entrada

Note que o vértice 5 tem grau nulo, ou seja, está isolado. Segue abaixo a MST gerada a partir do vértice 1 e a partir do vértice 5, respectivamente.

```
5
1 2 1.00
1 3 3.00
2 4 2.00
6.00
```

Figura 14: MST gerada a partir do vértice 1

```
5
0.00
```

Figura 15: MST gerada a partir do vértice 5

Como se pode ver, a MST gerada a partir do vértice isolado com grau nulo (Figura 15) não apresenta arestas.

3.3. Testes realizados com grafos disponibilizados no moodle da disciplina

3.3.1. Grafo 1

O grafo testado se encontra no arquivo data/GrafosMoodle/grafo_1.txt. O grafo tem tamanho 100 e 419 arestas. Pode-se calcular a mst e verificar um peso total de 336,00. Segue abaixo o menor caminho entre os vértices 1 e 10 e entre 1 e 100, respectivamente. A saída do programa está melhor detalhada no diretório “output”.

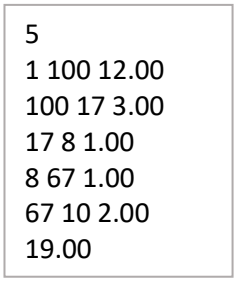


Figura 16: Menor caminho entre os vértices 1 e 10

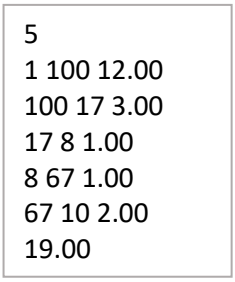


Figura 17: Menor caminho entre os vértice 1 e 100

3.3.2. Grafo 2

O grafo testado se encontra no arquivo data/GrafosMoodle/grafo_2.txt. O grafo tem tamanho 1000 e 337751 arestas. Pode-se calcular a mst e verificar um peso total de 999,00. Note que é quase o mesmo número de vértices; isso ocorreu devido ao fato de todos os pesos serem iguais a um. Segue abaixo o menor caminho entre os vértices 1 e 10, entre 1 e 100, e entre 1 e 1000, respectivamente. A saída do programa está melhor detalhada no diretório "output".

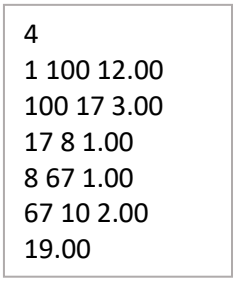


Figura 18: Menor caminho entre os vértices 1 e 10

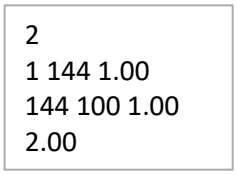


Figura 19: Menor caminho entre os vértices 1 e 100

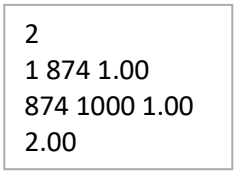


Figura 20: Menor caminho entre os vértices 1 e 1000

3.3.3. Grafo 3

O grafo testado se encontra no arquivo data/GrafosMoodle/grafo_3.txt. O grafo tem tamanho 10000 e 88558 arestas. Pode-se calcular a mst e verificar um peso total de 31947,00. Segue abaixo o menor caminho entre os vértices 1 e 10, entre 1 e 100, e entre 1 e 10000, respectivamente. A saída do programa está melhor detalhada no diretório "output".

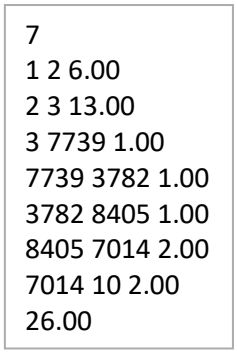


Figura 21: Menor caminho entre os vértices 1 e 10

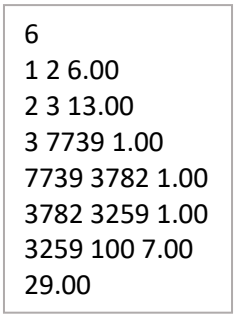


Figura 22: Menor caminho entre os vértices 1 e 100

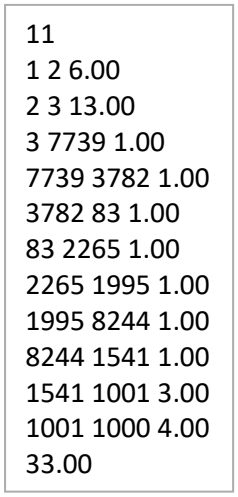


Figura 23: Menor caminho entre os vértices 1 e 1000

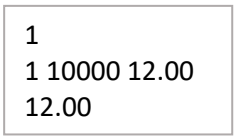


Figura 24: Menor caminho entre os vértices 1 e 10000

3.3.4. Grafo 4

O grafo testado se encontra no arquivo data/GrafosMoodle/grafo_4.txt. O grafo tem tamanho 50000 e 332089 arestas. Pode-se calcular a mst e verificar um peso total de 216236,00. Segue abaixo o menor caminho entre os vértices 1 e 10, entre 1 e 100, entre 1 e 1000 e entre 1 e 10000, respectivamente. A saída do programa está melhor detalhada no diretório “output”.

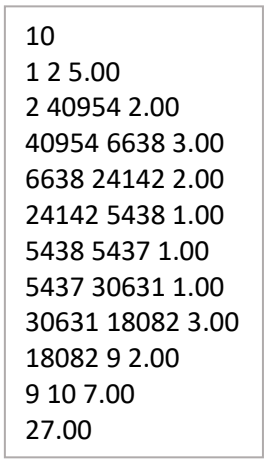


Figura 25: Menor caminho entre os vértices 1 e 10

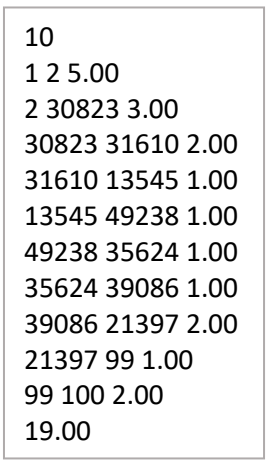


Figura 26: Menor caminho entre os vértices 1 e 100

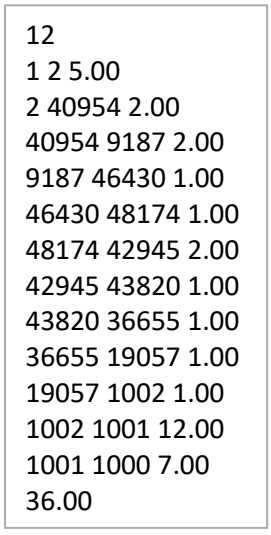


Figura 27: Menor caminho entre os vértices 1 e 1000

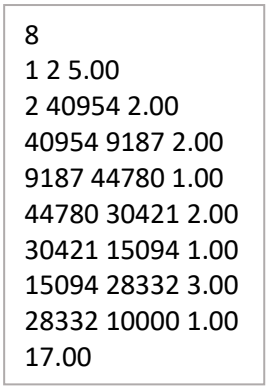


Figura 28: Menor caminho entre os vértices 1 e 10000

3.3.5. Grafo 5

O grafo testado se encontra no arquivo data/GrafosMoodle/grafo_5.txt. O grafo tem tamanho 100000 e 297881 arestas. Pode-se calcular a mst e verificar um peso total de 608677,00. Segue abaixo o menor caminho entre os vértices 1 e 10, entre 1 e 100, entre 1 e 1000 e entre 1 e 10000, respectivamente. A saída do programa está melhor detalhada no diretório “output”.

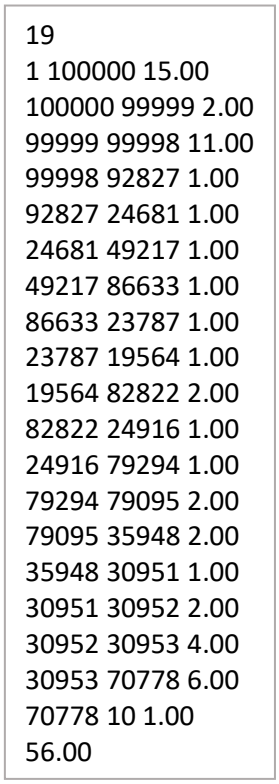


Figura 29: Menor caminho entre os vértices 1 e 10

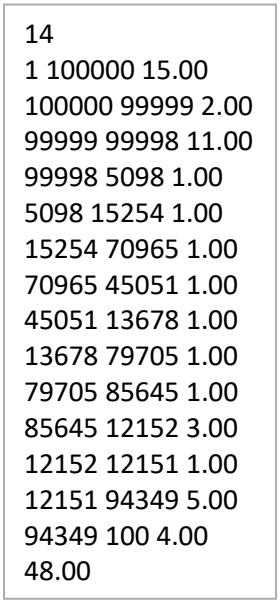


Figura 30: Menor caminho entre os vértices 1 e 100

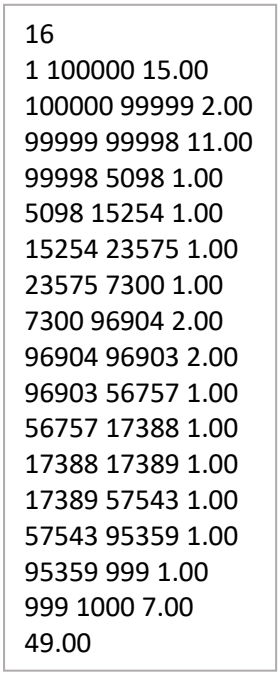


Figura 31: Menor caminho entre os vértices 1 e 1000

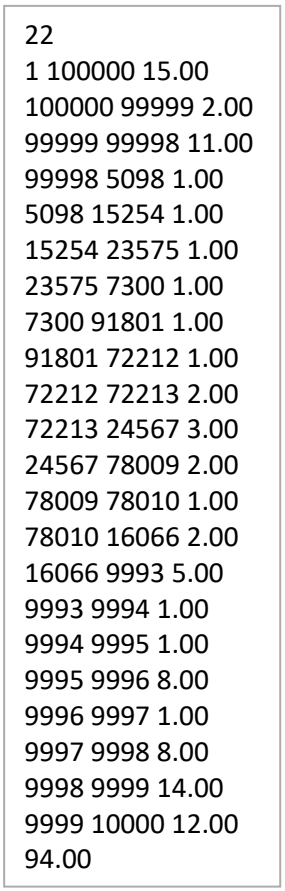


Figura 32: Menor caminho entre os vértices 1 e 10000

4. ANÁLISE DA COMPLEXIDADE

O estudo da complexidade será feito por funções. Depois será calculada a complexidade do programa por inteiro.

Segue abaixo uma tabela com as complexidades das funções.

Funções	Análise	Complexidade
Módulo de fluxo de dados e módulo de impressão		
recebedados ()	A leitura dos dados do grafo se repete E + 1 vezes.	O(E)
imprimeGrafo ()	Imprime as arestas de todos os nós do grafo, logo tem-se uma função que repete V vezes onde cada vez pode repetir até E vezes.	O(V*E)
imprimeAresta ()	Imprime uma única aresta.	O(1)
imprimeDistancia ()	Imprime uma lista de distâncias entre 1 vértice a todos os outros, logo é preciso realizar V repetições	O(V)
resultadoDijkstra ()	Imprime informações do estado do elevador. Não exige repetições.	O(1)
imprimeMST ()	Imprime os dados de tempo de execução. Como não depende do tamanho da entrada tem complexidade constante.	O(1)
Módulo de grafo		
getSize()	Função de complexidade constante.	O(1)
getEdge()	Função de complexidade constante.	O(1)
getGrau()	Função de complexidade constante.	O(1)
getPeso()	Função de complexidade constante.	O(1)
getPosicao()	Função de complexidade constante.	O(1)
setEdge()	Função de complexidade constante.	O(1)
setGrau()	Função de complexidade constante.	O(1)
nextGrau()	Função de complexidade constante.	O(1)
validaGrafo()	Função de complexidade constante.	O(1)

criaGrafo()	Função de cria e inicializa um grafo. Para isso precisa percorrer cada vértice para definir um grau inicial nulo, logo se repete V vezes.	O(V)
adicionaAresta()	Função de complexidade constante. Veja que o custo de inserção de uma aresta é O(1).	O(1)
add()	Função de complexidade constante.	O(1)
addAlocando()	Função de complexidade constante.	O(1)
addRealocando()	Função de complexidade constante.	O(1)
buscaVertice()	Busca o índice de vértice na lista de adjacência. No pior caso, os vértices possuem E arestas, logo esse é o tamanho da lista de adjacência. Como é uma busca linear, o pior caso é quando se precisa percorrer toda a lista.	O(E)
Módulo Dijkstra		
existeAberto()	Verifica se ainda existe vértice não visitado, logo percorre todos os vértices.	O(V)
inicializaD()	Realiza a inicialização dos vértices, por isso precisa percorrer todos.	O(V)
inicializaAberto()		
menorDist()	Busca o índice do vértice com menor distância na lista de distâncias de um vértice a todos os outros. Logo pode percorrer toda a lista que tem tamanho V.	O(V)
relaxa()	Realiza uma operação de relaxação que tem custo O(1).	O(1)
dijkstra()	A função dijkstra() seleciona vértice por vértice, o que implica em V repetições. Para cada uma dessas, percorre a lista de adjacência do vértice selecionado que pode ter tamanho até E. Logo a complexidade o algoritmo de Dijkstra é O(V*E).	O(V*E)
Módulo MST		
mstPrim()	O algoritmo de Prim seleciona vértice por vétice buscando o que tem valor menor, o que implica	O(V*E)

	em V repetições. Para cada uma dessas repetições, percorre a lista de adjacência do vértice selecionado que pode ter tamanho até E. Logo a complexidade desta implementação do algoritmo de Prim é O(V*E).	
minimum()	Percorre todos os vértices para retornar índice do vértice com menor valor.	O(V)

Conforme o quadro acima, verifica-se que o algoritmo de Dijkstra tem complexidade O(V*E) e a implementação do algoritmo de Prim tem compleximade O(V*E).

5. CONLUSÃO

A partir dos testes realizados foi possível verificar que os algoritmos de Dijkstra e de Prim foram implementados com sucesso de forma a resolver os problemas pelos quais eles foram propostos. Além disso, o tipo abstrato de dados Grafo foi implementa de forma correta e utilizável na representação de um grafo.

APÊNDICE 1 – MANUSCRITOS DOS GRAFOS UTILIZADOS NA ETAPA DE TESTES

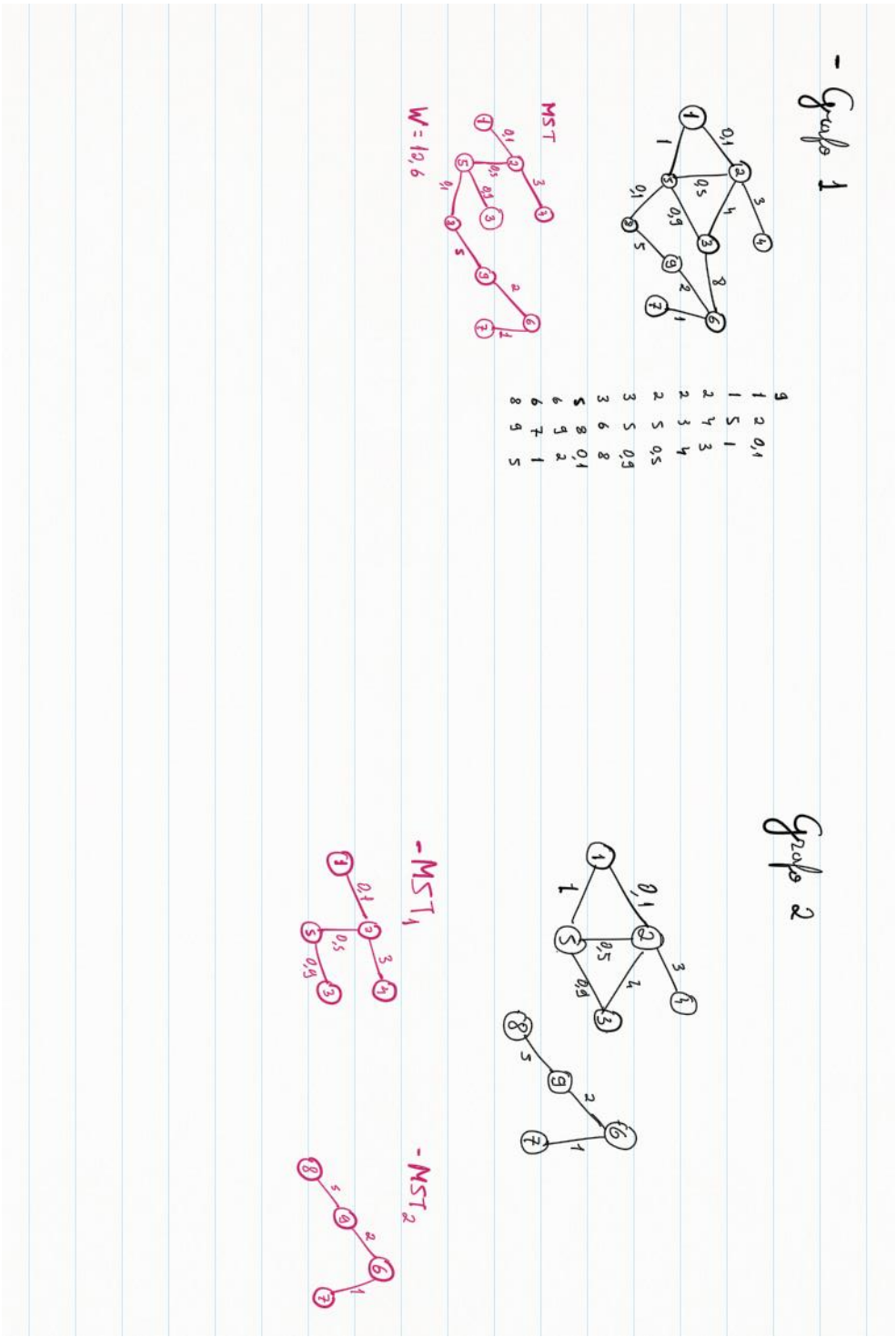
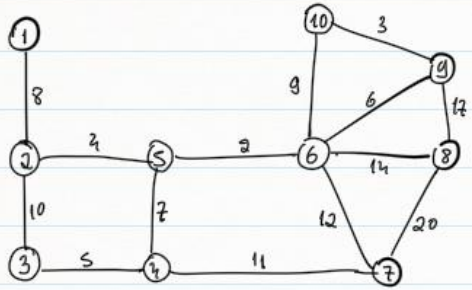


Figura 33: Anotações do grafo 1 e grafo 2 dos grafos selecionados

Grafo 3

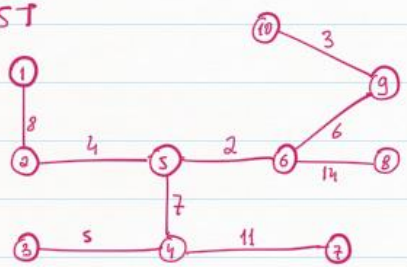


10
1 2 8
2 5 4
2 3 10
3 4 5
4 5 7
5 6 2
4 7 11
6 7 12
6 8 14
8 7 20

6 10 9
6 9 6
10 9 3
9 8 17

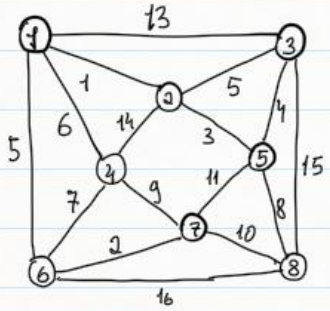
Menor caminho entre 1 e 10

MST

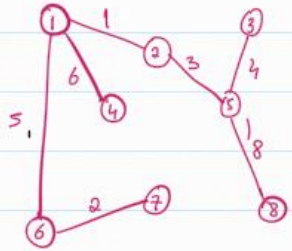


10
1 2 8
2 5 4
5 6 2
5 4 7
4 3 5
4 7 11
6 9 6
9 10 3
6 8 14

Grafo 4



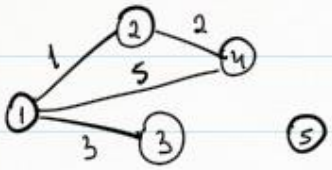
8
1 2 1
1 3 13
1 4 6
1 6 5
6 8 16
7 8 10
5 8 8
3 8 15
4 2 14
4 7 9
5 2 3
5 7 11
6 4 7
6 7 2
3 2 5
3 5 4



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

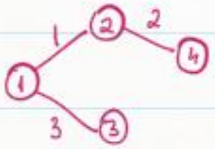
W = 29

Grafo 5 (Grafo não conectado)




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

MST



W = 6

Dijkstra (1, 4)

APÊNDICE 2 – IMPLEMENTAÇÃO DO CÓDIGO DO PRGRAMA

Arquivo principal (main.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "headers/grafo.h"
#include "headers/mst.h"
#include "headers/data.h"
#include "headers/dijkstra.h"
#include "headers/printter.h"
#include <limits.h>

int main(int argc, char* argv[]){
    //ENTRADA DE DADOS
    Grafo* grafoIn = recebeDados(argc, argv);
    if(argc == 2){
        //PROCESSAMENTO
        Aresta* mstArestas = mstPrim(grafoIn,0);//mst gerado com raiz 1
        float* d = dijkstra(grafoIn, 0);

        //SAIDA DE DADOS
        saidaPadrao(grafoIn,mstArestas,d,argv);
    }
    else if(!strcmp(argv[2],"1")){
        abreMenu(grafoIn);
    }
    else{
        printf("ERRO : Parece que houve uma tentativa de acesso ao menu, porém
o terceiro argumentO passado não é válido.\n");
        printf("Tente seguir o padrão: ./projGrafo arqGrafo.txt 1\n");
    }
    return 0;
}
```

Arquivo do módulo MST (mst.c)

```c
#include "headers/printter.h"
#include "headers/grafo.h"
#include "headers/mst.h"
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <limits.h>

#define INFINITO (float)INT_MAX/2.0f
```

```
/*
    Função que dado um grafo passado por argumento e o número do vértice raiz executa o algoritmo
    de Prim, retornando um vetor de Arestas que contém as arestas selecionadas pelo algoritmo.
*/
Aresta* mstPrim(Grafo* grafo, int r){
    int atribui = 1;
    Vertice* lista = (Vertice*)malloc(getSize(grafo)*sizeof(Vertice));
    Aresta* mstArestas = (Aresta*)malloc(getSize(grafo)*sizeof(Aresta));
    int vertAdj;
    float pesoMST;
    int i, u, indAdj;
    int count = 0 ;

    //inicializa os valores
    for (int i = 0; i<getSize(grafo); i++){
        lista[i] = grafo->vertice[i];
        lista[i].valor = INFINITO;
        lista[i].statusMST = 0;
        grafo->vertice[i].statusMST = 0;
        mstArestas[i].a = 0;
        mstArestas[i].b = 0;
        mstArestas[i].peso = 0.0;
    }

    //marca o valor do vertice raiz como 0, assim ele será selecionado na primeira chamada da função minimum()
    lista[r].valor = 0;

    //Repete-se selecionando todos os vértices do grafo.
    while(count < getSize(grafo)-1){
        atribui = 0;
        int u = minimum(lista,getSize(grafo));//seleciona o vértice que tem menor valor entre todos os outros ainda não visitados
        lista[u].statusMST = 1;

        //para o vértice selecionado, percorre toda a lista de adjacência atualizando o valor
        //dos vértices vizinhos quando forem maior que o peso da aresta
        for(indAdj=0; indAdj < getGrau(grafo,u); indAdj++){//Adjacentes de 0 até grau do vértice
            vertAdj = lista[u].listaAresta[indAdj].destino;
            if (lista[vertAdj].statusMST == 0 && getPeso(grafo,u,indAdj) < lista[vertAdj].valor){
                lista[vertAdj].valor = getPeso(grafo,u,indAdj);
                mstArestas[lista[u].listaAresta[indAdj].destino].a = u + 1;
                mstArestas[lista[u].listaAresta[indAdj].destino].b = lista[u].listaAresta[indAdj].destino + 1;
```

```
                mstArestas[lista[u].listaAresta[indAdj].destino].peso =
getPeso(grafo,u,indAdj);
                atribui = 1;
            }
        }
        count++;
    }
    return mstArestas;
}

/*Função que recebe uma lista de vertices e o tamanho do grafo
  Retorna o índice do vértice que possui menor valor
*/
int minimum(Vertice* lista, int V){
    float min = INFINITO;
    int minInd;;
    int i,indAdj;
    int existeMenor = 0;
    for(int indAdj = 0; indAdj < V; indAdj++){
        if(lista[indAdj].valor < min && lista[indAdj].statusMST == 0){
            min = lista[indAdj].valor;
            minInd = indAdj;
            existeMenor = 1;
        }
    }

    return minInd;
}
```

Arquivo do módulo Dijkstra (dijkstra.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
#include "headers/grafo.h"
#include "headers/dijkstra.h"
#define INFINITO (float)INT_MAX/2.0f

//---Dijkstra e Auxiliares---//

    /*Função que executa o algoritmo de Dijkstra.
      Recebe um grafo como argumento e o número do vértice a partir do qual
      Retorna uma lista das menores distâncias entre o vértice recebido como
argumento
      e os restantes.
    */
    float* dijkstra(Grafo* grafo, int s){
```

```
        int size = getSize(grafo);
        float* d = (float*)malloc(size*sizeof(float));
        bool aberto[size];
        incializaD(grafo, d, s);
        incializaAberto(aberto, size);
        while(existeAberto(aberto, size)){

            // Seleciona o vértice de menor distância até o momento
            int u = menorDist(grafo, aberto, d);

            // Fecha o vértice selecionado
            aberto[u] = false;

            // Percorre a lista de adjacencia de u relaxando as arestas
            for(int i = 0; i < getGrau(grafo, u) && getGrau(grafo, u) > 0;
i++){
                relaxa(grafo, d, u, grafo->vertice[u].listaAresta[i].destino,
i);
            }
        }
        return d;

    }

    /*
        Função que realiza uma operação de relaxação se a distância do vértice
v for
        maior que a distância do vértice u mais o peso da aresta
    */
    void relaxa(Grafo* grafo, float* distancia, int u, int v, int i){

        /*
            Verifica se o peso atual para alcanaçar a aresta v é maior que
            o peso da aresta u mais o peso de u para v
        */
        if(distancia[v] > distancia[u] + getPeso(grafo, u, i)){
            grafo->vertice[v].ant = u;
            distancia[v] = distancia[u] + getPeso(grafo, u, i);
        }
    }

    /*
        Retorna o índice do vertice com menor distância na lista d
     */
    int menorDist(Grafo* g, bool* aberto, float* d){
        int  i;
        // Descobre o primeiro aberto
        for(i=0; i < g->size; i++){
            if(aberto[i]){
```

```
                break;
            }
        }
        // Verifica se o grafo está todo fechado
        if(i == g->size){
            return -1;
        }
        int menor = i;

        // Seleciona o menor aberto
        for(i = i +1; i < g->size; i++){
            if(aberto[i] && (d[menor]) > d[i]){
                menor = i;
            }
        }
        return menor;
    }

    // Verifica se ainda existem vértices não visitados (aberto)
    bool existeAberto(bool* aberto, int size){
        for(int i = 0; i < size; i++){
            if(aberto[i]){
                return true;
            }
        }
        return false;
    }

//---To Inicializar---//

    // Inicializa lista de distâncias com valor infinito, exceto s
    void incializaD(Grafo* grafo, float* distancia, int s){

        for(int v = 0; v < grafo->size; v++){
            distancia[v] = INFINITO;
        }

        // s recebe zero pois ele é o ponto de partida
        distancia[s] =0.0f;
    }

    // Inicializa lista de aberto: default = true
    void incializaAberto(bool* aberto, int size){
        for(int i=0; i< size;i++){
            aberto[i] = true;
        }
    }
```

Arquivo do módulo de dados (data.c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "headers/grafo.h"
#include "headers/dijkstra.h"
#include "headers/mst.h"
#include "headers/printter.h"
#include "headers/data.h"

Grafo* recebeDados(int argc, char **argv){
    int numVertices;
    int aresta1,aresta2;
    double weight;
    int linha = 1;
    Grafo* grafo  = (Grafo*)malloc(sizeof(Grafo));

    FILE *arqGrafo = fopen(argv[1], "r");

    if(arqGrafo == NULL){
        printf("\nERRO 2: Não foi possível abrir o arquivo de grafo\n");
        exit(1);
    }
    else{
        fscanf(arqGrafo, "%d\n",&numVertices);
        grafo = criaGrafo(numVertices);

        while(fscanf(arqGrafo, "%d %d %lf\n", &aresta1, &aresta2,
&weight)!=EOF){
            linha++;

            adicionaAresta(grafo,aresta1-1,aresta2-1,weight);
            if(weight < 0){
                printf("ERRO 3: peso de aresta negativo\n");
                printf("Este algoritmo não funciona com valores negativos para
pesos de arestas,\npor favor altere a linha %d do arquivo
%s\n",linha,argv[1]);
                exit(1);
            }
            if(aresta1 == aresta2){
                printf("AVISO 1: ciclos não serão considerados neste
programa.\n");
                printf("Existe um ciclo na linha %d do arquivo
%s\n",linha,argv[1]);
            }
            //Guardar os valores na estrutura
        }
        setEdge(grafo,linha - 1);
```

```
    }
    fclose(arqGrafo);
    if(getEdge(grafo)==0){
        printf("ERRO 4: Nenhuma das arestas inseridas são válidas. Reveja o
arquivo de entrada.\n");
        exit(1);
    }
    return grafo;
}

/*
  Função que exibe a saida padrão, ou seja, para um dado grafo de entrada
executa as operações:
    - menor caminho entre 1 e 10 (se existir)
    - menor caminho entre 1 e 100 (se existir)
    - menor caminho entre 1 e 1000 (se existir)
    - menor caminho entre 1 e 10000 (se existir)
    - calcula MST
*/
void saidaPadrao(Grafo* grafoIn, Aresta* mstArestas, float* d, char* argv[]){
    FILE *mstOutput, *dijkstraOutput;
    char nome_arqMST[20] = "output/mst";
    char nome_arqDij[20] = "output/dijkstra";
    strcat(nome_arqMST,".txt");
    strcat(nome_arqDij,".txt");
    mstOutput = fopen(nome_arqMST,"w");
    dijkstraOutput = fopen(nome_arqDij,"w");

    if(getSize(grafoIn) >= 10){
        resultadoDijkstra(grafoIn,d, 0, 9,dijkstraOutput);//menor caminho
entre o vértice 1 e 10
        fprintf(dijkstraOutput,"\r\n");
        printf("\n");
    }
    if(getSize(grafoIn) >= 100){
        resultadoDijkstra(grafoIn,d, 0, 99,dijkstraOutput);//menor caminho
entre o vértice 1 e 100
        fprintf(dijkstraOutput,"\r\n");
        printf("\n");
    }
    if(getSize(grafoIn) >= 1000){
        resultadoDijkstra(grafoIn,d, 0, 999,dijkstraOutput);//menor caminho
entre o vértice 1 e 1000
        fprintf(dijkstraOutput,"\r\n");
        printf("\n");
    }
    if(getSize(grafoIn) >= 10000){
        resultadoDijkstra(grafoIn,d, 0, 9999,dijkstraOutput);//menor caminho
entre o vértice 1 e 10000
```

```
            fprintf(dijkstraOutput,"\r\n");
            printf("\n");
        }

        imprimeMST(grafoIn,mstArestas,mstOutput);
        fprintf(mstOutput,"\r\n");

        fclose(mstOutput);
        fclose(dijkstraOutput);
        printf("\n");
}

/*
    Função que exibe um menu de operações:
      1 - MST
      2 - Menor caminho entre dois vértices
      0 - Sair
*/
void abreMenu(Grafo* grafoIn){
    int opcao = -1;
    int raiz;
    int verticeOrigem, verticeDestino;
    float* d;
    FILE *mstOutput, *dijkstraOutput;
    char nome_arqMST[20] = "output/mst";
    char nome_arqDij[20] = "output/dijkstra";

    printf("Você é livre para realizar operações sobre o grafo\nMENU:\n");
    do{
        printf("1 - MST\n");
        printf("2 - Menor caminho entre dois vértices\n");
        printf("0 - Sair\n");
        scanf("%d",&opcao);

        switch (opcao){
            case 0:
                break;
            case 1:
                strcat(nome_arqMST,".txt");
                mstOutput = fopen(nome_arqMST,"w");
                printf("\nO grafo tem tamanho %d:\n",getSize(grafoIn));
                printf("Digite o número do vértice raiz a partir do qual a MST
é gerada:\n");
                printf("OBS.: Deve estar no intervalo fechado
[1,%d]\n",getSize(grafoIn));
                scanf("%d",&raiz);
                if(1<= raiz && raiz <= getSize(grafoIn)){
                    Aresta* mstArestas = mstPrim(grafoIn,raiz - 1);
                    imprimeMST(grafoIn, mstArestas,mstOutput);
```

```
                    fprintf(mstOutput,"\r\n");
                    printf("\n");

                }
                else{
                    printf("ERRO: Vértice raiz inválido. Para o grafo inserido
de tamanho %d insira um vértice raiz \n com valor entre 1 e
%d\n",getSize(grafoIn),getSize(grafoIn));
                }
                fclose(mstOutput);
                break;

            case 2:
                strcat(nome_arqDij,".txt");
                dijkstraOutput = fopen(nome_arqDij,"w");
                printf("\nO grafo tem tamanho %d:\n",getSize(grafoIn));
                printf("Digite os números do vértice origem e do vértice
destino:\n");
                printf("OBS.: Ambos os vértices devem estar no intervalo
fechado [1,%d]\n",getSize(grafoIn));
                printf("Insira o vértice origem: ");
                scanf("%d",&verticeOrigem);
                printf("Insira o vértice destino: ");
                scanf("%d",&verticeDestino);

                if(1<= verticeOrigem && verticeOrigem <= getSize(grafoIn) ||
1<= verticeDestino && verticeDestino <= getSize(grafoIn)){
                    d = dijkstra(grafoIn, verticeOrigem - 1);
                    resultadoDijkstra(grafoIn,d, verticeOrigem - 1,
verticeDestino - 1,dijkstraOutput);//menor caminho entre o vértice 1 e 10000
                    fprintf(dijkstraOutput,"\r\n");
                }
                else{
                    printf("Algum dos vértices são inválidos. Para o grafo
inserido de tamanho %d insira vértices \n com valor entre 1 e
%d\n",getSize(grafoIn),getSize(grafoIn));
                }
                fclose(dijkstraOutput);
                break;

            default:
                printf("Opção inválida.\n");
                break;
        }

    }while(opcao != 0);

    return;
}
```

Arquivo do módulo de impressão (printter.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "headers/printter.h"
#include "headers/mst.h"
#include "headers/grafo.h"
#define INFINITO INT_MAX/2.0f

//---Printters (Hp, é claro)---//

    void imprimeGrafo(Grafo* grafo){
        FILE* null;
        for (int i = 0; i < grafo->size; i++) {
            for(int j = 0; j < getGrau(grafo, i); j++){
                imprimeAresta(grafo, i, j,null);
            }
            printf("\n");
        }
    }

    void imprimeAresta(Grafo* grafo, int a, int n, FILE *dijkstra){
        if (n >= getGrau(grafo, a)) {
            printf("Aresta Inexixtente!\n");
            exit(1);
        }

        printf("%d ", a + 1);
        printf("%d ",grafo->vertice[a].listaAresta[n].destino + 1);
        printf("%.2f\n",grafo->vertice[a].listaAresta[n].peso);
        fprintf(dijkstra,"%d %d %.2f\r\n", a+1,grafo-
>vertice[a].listaAresta[n].destino + 1,grafo->vertice[a].listaAresta[n].peso);

        if(grafo->vertice[a].ant + 1 == 0){
            printf("vertice inicial\n");
            return;
        }
    }

    void imprimeDistancia(float* d, int size){
        for(int i = 0; i < size; i++){
            if(d[i] == INFINITO){
                printf("%d: Infinito\n", i);
            } else {
                printf(("%d: %.2f\n"),i,d[i]);
            }
        }
    }
```

```
    void resultadoDijkstra(Grafo* grafo, float* d, int origem, int destino,
FILE* dijkstra){

        if (d[destino] == INFINITO){
            printf("O vértice %d não é alcançável a partir de %d\n", destino +
1, origem + 1);
            return;
        }

        int size = getSize(grafo);
        int i, j = size - 1;
        int back[size];
        float peso = 0.0f;

        for(int cont = 0; cont < size; cont++){
            back[cont] = -1;
        }

        back[j] = destino;
        j--;
        for(i = destino;  i != origem; ){
            i = grafo->vertice[i].ant;
            back[j] = i;
            j --;
        }
        int numeroSaltos = size - j - 2;
        printf("%d\n",numeroSaltos);
        fprintf(dijkstra,"%d\r\n", numeroSaltos);
        for(j = j + 1; j < size - 1; j++){
            int temp = buscaVerticeAdj(grafo, back[j], back[j+1]);
            peso += grafo->vertice[back[j]].listaAresta[temp].peso;
            imprimeAresta(grafo, back[j], temp, dijkstra);
        }
        fprintf(dijkstra,"%.2f\r\n", peso);
        printf("%.2f\n",peso);
    }

    void imprimeMST(Grafo* grafo, Aresta* mstArestas,FILE* mst){
        float pesoMST = 0;
        printf("%d\n",getSize(grafo));
        fprintf(mst,"%d\r\n", getSize(grafo));
        for(int u = 0;u < getSize(grafo); u++){
            pesoMST += mstArestas[u].peso;
            if(mstArestas[u].peso != 0.0){
                fprintf(mst,"%d %d
%.2f\r\n",mstArestas[u].a,mstArestas[u].b,mstArestas[u].peso);
                printf("%d %d
%.2f\n",mstArestas[u].a,mstArestas[u].b,mstArestas[u].peso);
```

```
        }
    }
    printf("%.2f\n",pesoMST);
    fprintf(mst,"%.2f\r\n", pesoMST);
}
```