Documentação do Projeto de Programação 2

Minimal Spanning Tree

Alunos: Bruna Azambuja e Ricardo Santos
Matrículas: 18/0014153 e 18/0027263
Disciplina Estrutura de Dados – turma E (2/2018)
Professor George Teodoro

S u m á r i o

1. Proposta do Projeto

Uma distribuidora de energia elétrica surgiu com um novo projeto para levar luz para vilarejos no interior do país. Para isso, é calculado o custo necessário para levantar postes entre todas as casas da cidade, considerando sempre aquele em que apresenta o menor investimento.

Contratou-se então um programador para encontrar a distribuição de conexões entre casas que desse o menor custo possível para a empresa, de forma automatizada; Assim, o projeto poderia se estender para qualquer cidade do país.

Para resolver esse problema, foi preciso implementar um algortimo que procura em grafos a árvore de extensão mínima - MST, que recebe como entrada o número de casas e a configuração das conexões entre as casas da cidade, com seus custos; e deve retornar para o usuário as arestas da MST com seus respectivos pesos, quanto foi economizado – diferença entre uma cidade totalmente conexa e a MST encontrada, e se a solução encontrada é única ou não – um determinado grafo pode ter mais de uma árvore com peso mínimo.

2. Como o algoritmo funciona ?

O alogoritmo contará com um parâmetro de entrada, como já foi comentado anteriormente. A entrada será o número total de vértices na primeira linha e em seguida uma matriz de adjacência de um grafo que representa quais casas (vértices) têm conexão (arestas) e qual seu peso. Terá o seguinte formato:

Arquivo de Entrada -

N – número total de vértices/casas

	X1	X2	X3	...	Xn
X1	X1-X1	X1-X2	X1-X3	...	X1-Xn
X2	X2-X1	X2-X2	X2-X3	...	X2-Xn
X3	X3-X1	X3-X2	X3-X3	...	X3-Xn
...			...		
Xn	Xn-X1	Xn-X1	Xn-X1	...	peso da conexão Xn-Xn

A saída do programa será printada num arquivo criado pelo próprio programa, que terá o nome de Exit_Matrix_MST, que contará com as arestas que compõe a MST, de forma ordenada por índice do menor vértice, e terá o seguinte formato:

Arquivo de Saída -

```
Y1 Y? - aresta do vértice y1 ao vértice ?
Y2 Y? - aresta do vértice y2 ao vértice ?
...
Y? Yn – aresta do vértice ? ao vértice n
```

---

onde Y1 < Y2 < … < Yn

Sabendo que além do arquivo de saída com a configuração da MST, será também printada uma resposta no próprio terminal, com os seguintes dados:

Saída do Terminal -

```
W1 – custo da MST;
W2 – custo economizado vs malha totalmente conexa (total – MST);
Informação Booleana – MST única ou não.
```

Caso a informação booleana da saída seja falsa – ou seja, caso MST não seja única, o programa irá retornar também outro arquivo de saída, com o nome Second_MST que conterá a segunda MST caso seja existente, e terá o mesmo formato do primeiro arquivo – Exit_Matrix_MST.

Com o parâmetro especificado, a execução do programa deverá se dar com a seguinte linha:

./mst.out arquivo_entrada

A lógica do programa se dará pelo já conhecido algoritmo de Prim, que será melhor explicado na próxima seção.

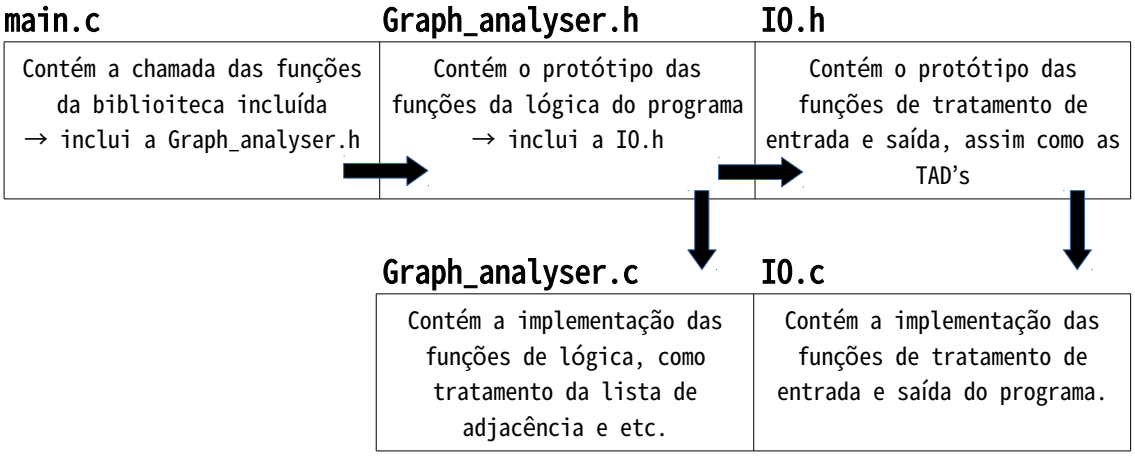
3. Como é dada a implementação ?

3.1. Separação dos módulos e sua inter-dependência

O programa foi separado em três módulos, sendo um para o programa principal (.c) – que será chamado de main, e dois (.c) que contém o protótipo das funções utilizadas na main, com suas respectivas bibliotecas (.h) – que serão chamados de IO (in/out) e Graph_analyser.

O diagrama das principais funções de cada módulo e a interdependência entre eles se encontra abaixo:

→ Desenho esquemático seção 3 – Figura 1: Módulos.



3.2. Descrição das Estruturas de Dados utilizadas

As TAD's utilizadas foram organizadas da seguinte forma:

- struct info: {
  int source, weight, dest; → *será usada para guardar as informações de uma aresta.*
  }
- struct edge: {
  struct edge* next; → *apontará para a próxima estrutura de aresta, que por sua vez terá o campo próximo e o campo dados .*
  info data; → *conterá os dados de uma struct do tipo info.*
  }
- struct list: {

edge* first; → *apontará para a primeira struct do tipo aresta da lista dinâmica.*

edge* last; → *apontará para a última struct do tipo aresta da lista dinâmica.*

struct list* parent; → *apontará para onde o vértice atual criou a conexão.*

float key; → *será a chave dos vértices, iniciada com infinito.*

int num_connections; → *informará quantas conexões o vértice em questão possui.*

}

- struct root{

int data; → *guardará o número identificador do vértice.*

struct root* left; →*apontará para o nó raiz da subárvore à esquerda.*

struct root* right; →*apontará para o nó raiz da subárvore à direita.*

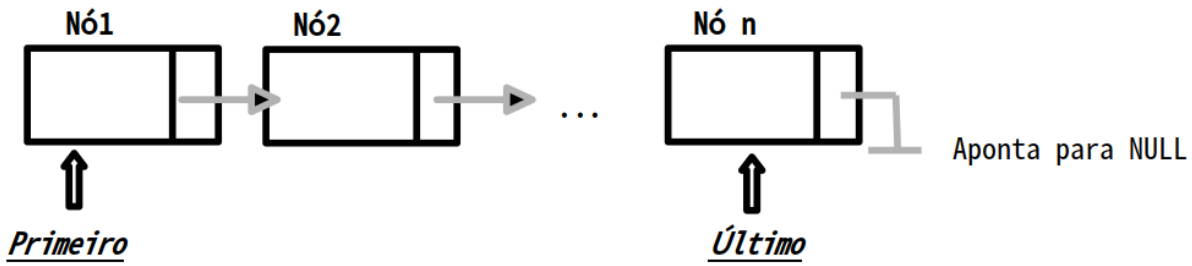
}

3.3. Implementando as TAD's

As TAD's descritas no tópico anterior foram implementadas de diferentes formas, para diferentes funções, que explicaremos melhor nessa seção.

A struct *edge* possui o formato do que chamaremos de Nó de uma lista dinâmica. Sendo assim, esse Nó possui um campo de *dados*, que será preenchido, nesse caso, com uma struct *info* – informações de cada aresta tais como vértices ligados por ela e seu peso; e o campo *próximo*, que armazena um ponteiro para outro Nó, com o mesmo formato do primeiro. Com a implementação dessa forma, conseguimos fazer uma lista dinâmica, ou seja, uma lista de Nós, onde cada um aponta para seu sucessor, sem tamanho fixo. Quando chegamos ao último da lista, seu apontador aponta para NULL.

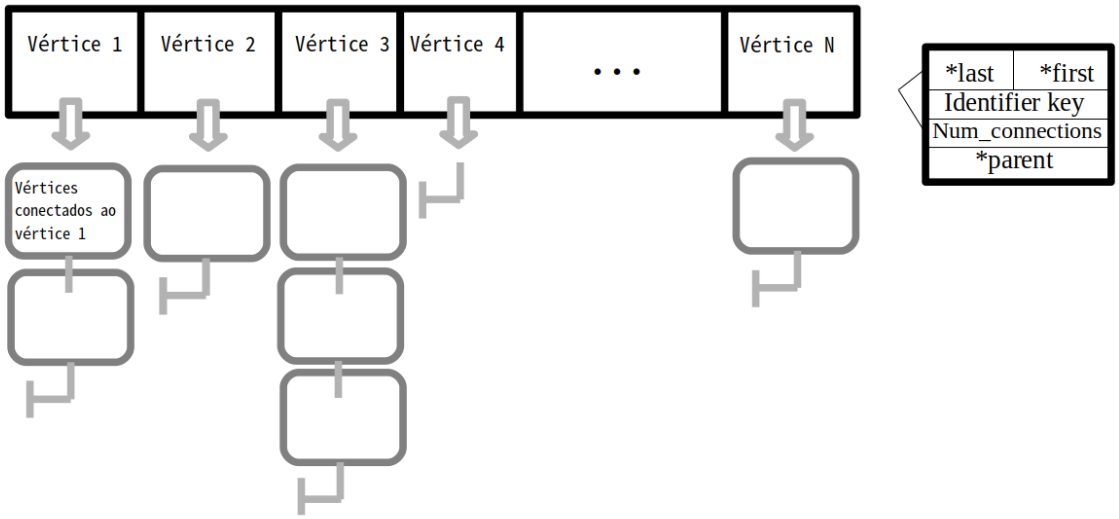
→ Desenho esquemático seção 3 – Figura 2: Lista Dinâmica.



A struct *list*, por sua vez, armazena cinco informações. Nesse caso, ponteiros para o primeiro e o último Nó da respetiva lista; bem como um apontador para o *parent* do vértice atual – o vértice responsável por incluir o atual na MST; um número float *key*, que será inicializada todos como infinito, e será sempre atualizada com o peso necessário para chegar do vértice pai ao vértice atual; e por fim, um inteiro *num_connections*, que representará o número de arestas ligadas ao vértice atual.

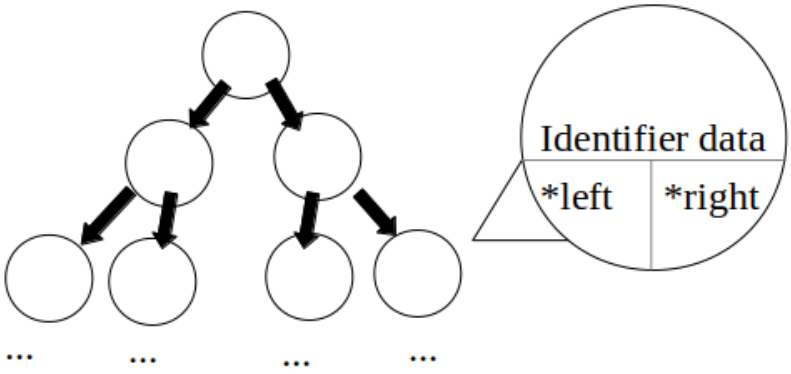
Essa lista dinâmica representada acima foi utilizada no programa como uma lista de adjacência do grafo de entrada. Ou seja, um vetor estático onde cada posição aponta para uma lista dinâmica, e seu íncide representa um vértice. Representada a seguir:

→ Desenho esquemático seção 3 – Figura 3: Lista de Adjacência.



Por fim, a struct *root* possui o formato utilizado numa árvore binária; que guardará o endereço do nó raiz da subárvore à esquerda e à direita, bem como um número inteiro *data* identificador do vértice atual. Será constituída apenas pelos vértices já inseridos na MST.

→ Desenho esquemático seção 3 – Figura 4: Árvore Binária.



4. Listagem do programa fonte

Depois de explicado como cada Estrutura de Dados se comporta, será mostrado como essas estruturas foram utilizadas na aplicação da lógica do programa, passo a passo, descrevendo, função por função, o relevante do programa principal.

4.1. Tratamento das entradas

Após a inicialização de todas as variáveis e estruturas que irão ser usadas, fez-se o tratamento de entradas do programa. Lê-se o arquivo de entrada, linha por linha da matriz de adjacência, e, criado um vetor onde cada posição é um apontador para uma lista dinâmica, foi preenchendo-se a lista de adjacência.

Tal lista foi preenchida como já foi esquematizado anteriormente.

→ Pseudocódigo seção 4 – número 1.

```
while read the file until EOF
        if weight != 0
                insert the edge on the adjacency list
```

4.2. Tratamento da lógica

Dentro da função *PrimAlg* – Algoritmo de Prim, inicia-se um loop *(while)* com uma variável *found_edges* – que representa a quantidade de arestas encontradas na MST, inicializada com zero. Para cada rodada no loop, ou seja, para cada aresta encontrada e inserida na MST, essa variável é incrementada. O loop irá rodar até que o número de arestas encontradas seja igual ao (número de vértices – 1) → propriedade da MST.

1. Antes de tudo, atualiza-se a chave do vértice em que será iniciada a procura – *(min_vertice),* que será no nosso caso o vértice de ínidice [0]. Seu significado será a distância necessária para partir do vértice pai ao vértice atual, ou seja, como começamos desse vértice, a chave é atualizada para zero;

No loop, serão feitas as seguintes operações, nessa ordem:

2. Insere-se o vértice atual na árvore binária *(BinarySearchTree)* – lembrando que a árvore binária de busca é organizada de forma que os valores maiores que o pai estão na subárvore à direita, e menores à esquerda;

3. Para cada conexão entre o vértice atual e algum outro, é verificado se o destino da conexão já está na MST (ou seja, já está na árvore), e se o peso é menor que a chave do vértice de destino. Caso o vértice de destino não esteja na MST ainda e o peso da aresta para chegar nele seja menor que sua chave, atualiza-se sua chave;

→ Pseudocódigo seção 4 – número 2.

```
for each connection between source vertex and dest
        if dest vertex is not on MST yet && dest vertex key > weight of the edge
                update dest vertex key = weight of the edge
```

4. Encontra primeiro vértice válido – ou seja, não está na MST ainda;

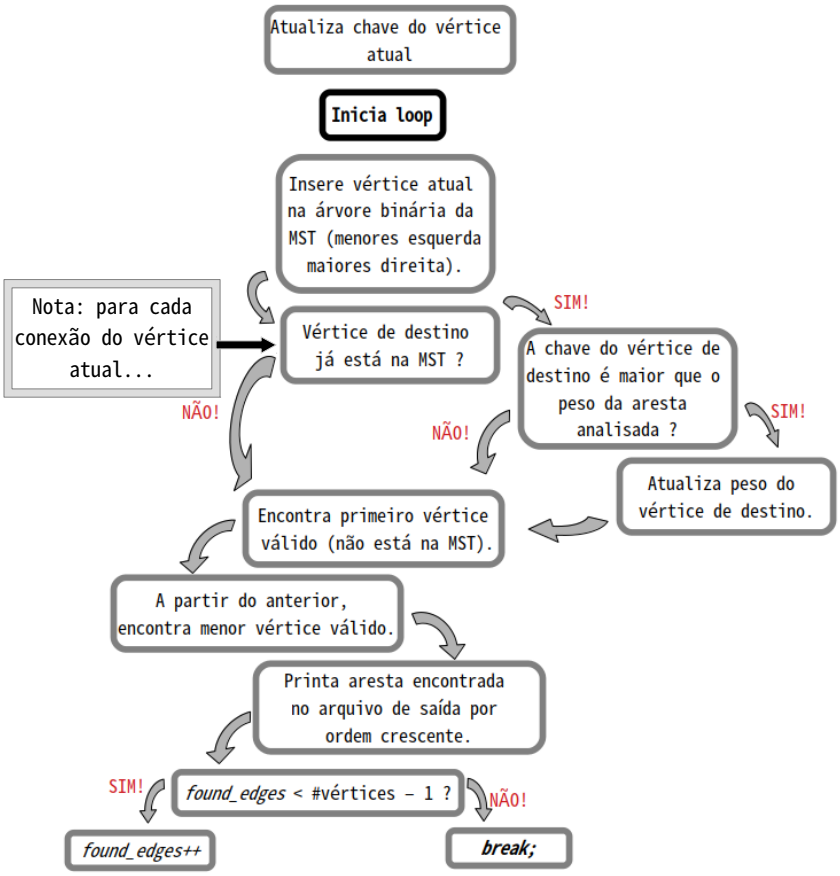
5. Encontra, a partir do primeiro válido, o vértice com o menor chave que também seja válido e atualiza o valor do *min_vertice* – que será usado como ponto de partida do próximo loop do algoritmo;

6. Atualiza-se o valor da variável *min_weight* que representa o peso total da MST, somando com o peso da aresta que acabamos de inserir na árvore binária, e incrementa-se a variável *found_edges*;

7. Printa no arquivo a aresta que acabamos de descobrir, por ordem crescente dos índices dos vértices – tanto da esquerda pra direita, quanto de cima para baixo;

8. Roda mais um loop…

→ Desenho esquemático seção 4 – Figura 5: Prim Algorithm.



4.3. Como foi encontrada a segunda MST ?

Agora que já foi explicado como uma MST foi encontrada, será explicitado como o programa analisa o caso da MST ser única ou não, e se não for, retornar também mais uma.

Como já citado anteriormente, aplicamos o Prim Algorithm para o vértice de índice [0] na última seção. Agora, para checar se a MST é única, deve-se aplicar o mesmo algoritmo a partir de outros vértices, verificando sempre se a MST encontrada para esse vértice é igual à encontrada do primeiro.

Aplicaremos o algoritmo até que: ou passemos por todos os vértices – MST é única e sempre encontramos a mesma, a partir de vértices diferentes; ou se encontre uma MST diferente da primeira. Nesse último caso, o loop é interrompido e o programa printa a segunda MST num outro arquivo de saída chamado Second_MST.

→ Pseudocódigo seção 4 – número 3.

```
        rewind(file pointer)
        for i: 1 → number of vertices
                free(Binary Tree)
                update all the keys to ∞
                PrimAlgorithm( adjacency_list [i] )
                if second_MST
                        break;
```

4.4. Cálculo do relatório de saída

Os cálculos dos dados pedidos no relatório foram dados por:

1. peso da MST = Somatório do peso de todas as arestas que compõem a MST;

2. economia da MST vs Malha Totalmente Conexa = (peso total de todas as arestas do grafo original) – (peso da MST).

Após o algoritmo finalizado, MST'(s) encontradas e todos os dados calculados, desaloca-se o espaço usado no programa. Função *freeTree* e *freeList* – percorre toda a árvore binária de busca e a lista adjacente, respectivamente, dando free(nó por nó). Bem como no array de ponteiros - que foi alocado dinamicamente.

5. Como é dada a compilação ?

Para a compilação, foi criado um arquivo Make File, que trata de todo o processo de compilação necessário para o funcionamento do programa. Nesse arquivo, incluiu-se a compilação tanto da IO.c quanto da graph_analyser.c, bem como a do programa principal – main.c.

Esse método de compilação se dá muito eficiente pois, além de reduzir o número de comandos necessários para compilar o programa inteiro, ele otimiza o processo, verificando a data de compilação de cada módulo. Ou seja, cada .c só será compilada novamente se alguma modificação foi feita desde a última vez.

Para utilizar o Make File basta digitar o comando "make" no terminal, e a saída será a compilação dos objetos que foram modificados.

6. Estudo da complexidade

Para termos uma noção melhor da eficiência do algoritmo, foi feita uma análise da complexidade de cada função das duas bibliotecas incluídas. Essa análise será comentada abaixo, bem como uma breve descrição do que a função associada faz, os parâmetros recebidos por ela e seu tipo.

Nesta seção, consideraremos 'E' o número de arestas *(edges)* e 'V' o número de vértices *(vertex)*. **Sabendo de antemão que para efeito de cálculo de complexidade, E é da ordem de V^2.**

1. `void Implementer(int num_vertices, vertice **adjacency_matrix, FILE *fp, int *total_weight);`

Recebe o ponteiro do arquivo de entrada, bem como: o número total de vértices, o endereço de uma variável que guardará o peso total da MST e o 'esqueleto' da lista de adjacência – array de listas vazias com [num_vertices] posições. A função irá ler a matriz de configuração do grafo e irá preencher a lista de adjacência, chamando a função *insert_list*.

Como a complexidade da função *insert_list* é da ordem de O(1) - como será analisado com mais detalhes adiante; e a função *implementer* a chama para cada aresta que tem o peso diferente de zero na matriz de adjacência, ou seja, da ordem de E, portanto no pior caso será da ordem de E. Complexidade O(E).

2. `root* NewNode(int data);`

Recebe um inteiro que representa o índice do vértice que será inserido na árvore binária, portanto, na MST. Retorna o ponteiro do nó que acabamos de criar. Complexidade: é uma função com 4 ações, traz uma complexidade +4 cada vez que é chamada - O(1).

3. `int Search_BSTtree(root *Bin_tree ,int data);`

Recebe o ponteiro da árvore binária de busca, e o índice do vértice que estamos procurando. Retorna uma informação booleana que nos diz se o vértice procurado está ou não na MST. É uma função recursiva que é chamada para a subárvore à esquerda e depois à subárvore à direita, verificando sempre se o nó atual é o vértice que procurado. Complexidade: é uma função recursiva de busca em uma árvore, logo no pior caso será da ordem de V, porém em média apresenta O(log V).

4. `void PrimAlg(vertice **adjacency_list, root *BST_tree, vertice *min_vertice, int num_vertices, FILE *fp, int *min_weight, int_list *exit_list, int FirstOrSecond);`

A função recebe o ponteiro da lista de adjacência, o ponteiro da árvore binária, o vértice pelo qual iremos começar a busca pela MST, o número total de vértices, o ponteiro do arquivo em que será printada a saída, o endereço de uma variável que conterá o peso da MST, a lista dinâmica que conterá as arestas da MST, e um parâmetro para diferenciar se o algoritmo precisa achar a primeira MST, ou a segunda.

A função e sua lógica já foi explicada em seções anteriores, e possui complexidade de acordo com os dois loops internos – ambos com pior caso na ordem de V iterações, e com as funções chamadas dentro desses loops, que gera um total de $O(V^2 \log V)$ para grafos mais densos, e para grafos menos densos algo na ordem de $O(V^2)$.

5. `void freeTree(root *BST_tree);`

Recebe o ponteiro da raiz da árvore binária. Essa função é chamada recursivamente para as subárvores à esquerda e à direita, para ir desalocando, nó por nó, a árvore. Apresentando, portanto, complexidade O(V).

6. `void UpdateKeys(vertice **adjacency_list, int num_vertices);`

Recebe o ponteiro da lista de adjacência, e o número total de vértices. A função atualiza a chave de todos os vértices para infinito, bem como o ponteiro *parent* para NULL. Complexidade: a função passa por todos os vértices, portanto apresenta O(V).

7. `void memoryAllocator(int num_vertices, vertice **adjecency_list);`

Recebe o ponteiro da lista de adjacência, bem como o número total de vértices. A função cria uma lista dinâmica vazia para cada posição da lista de adjacência, fazendo com que todos os ponteiros apontem para zero e que suas chaves sejam inicializadas com infinito. Complexidade: a função percorre toda a lista de adjacência, fazendo um total de 6 operações para cada posição. Portanto O(V).

8. `void Insert_Tree(root **node, int data);`

Recebe o ponteiro da raiz da árvore binária, bem como o índice do vértice que deve ser inserido. É uma função recursiva chamada para a subárvore à esquerda – caso o índice seja menor que a raiz da subárvore; e à direita – caso o índice seja maior que a raiz da subárvore. Complexidade no pior caso é da ordem de V, porém em geral, busca em árvore binária apresenta complexidade de O(logV).

9. `void freeList(vertice **adjacency_list, int num_vertices);`

Recebe o ponteiro da lista de adjacência e o número total de vértices. A função percorre, para todas as posições do array[num_vertices], a lista dinâmica até o final, dando free nó por nó. Complexidade: O(E).

10. `void Insert_list(vertice **adjacency_list, int source, int dest, int weight);`

Recebe o ponteiro da lista de adjacência, dois inteiros que representam quais vértices serão inseridos na lista de adjacência, e seu peso. Como o nó é inserido no fim da lista, e sua estrutura tem um ponteiro direto para o fim, possui portanto Complexidade: O(1).

11. `void freeExit(int_list *exit_list);`

Recebe o ponteiro da lista de saída, que contém as arestas da MST. Percorre a lista inteira, dando free nó por nó. Complexidade: como a lista tem |V| - 1 arestas, portanto O(V).

12. `void InsertExitOrganized(int_list *exit_list, int source, int dest);`

Recebe o ponteiro da lista de saída, bem como os vértices que são ligados pela aresta que será inserida na lista. Como é uma inserção ordenada, temos que procurar a posição em que ficará a nova aresta, portanto Complexidade O(V).

13. `int SearchForPair(int source, int dest, int_list *exit_list);`

Recebe o ponteiro da lista de saída, bem como os vértices cuja aresta estamos procurando. Percorre a lista inteira, procurando pelo par de vértices, portanto Complexidade O(V).

14. `void Tranverse_and_print(int_list *exit_list, FILE *fp);`

Recebe o ponteiro da lista de saída, bem como o do arquivo de saída. A função percorre toda a lista de arestas, printando no arquivo, uma por uma. Portando apresenta complexidade de O(V).

15. Funções clock:

A função é chamada 2 vezes e possui complexidade O(1);

16. Complexidade do programa inteiro:

A complexidade geral do programa, portanto, possui uma alta dependência do fato de haver ou não uma segunda MST e, no caso de haver, também depende de a partir de qual vértice pode ser encontrada.

Por exemplo, há o caso em que o algoritmo é aplicado nos dois primeiros vértices e ja serem encontradas duas MST's diferentes e o programa acabar. Porém, também há casos em que uma segunda MST só é encontrada passando

pelo úlitmo vértice, ou ainda, percorrer todos os vértices e não encontrar uma segunda MST.

Logo, tudo depende. Mas em geral, a complexidade do programa pode ser calculada pela maior complexidade – ou seja, a da função *PrimAlg*, dentro do loop que roda por todos os vértices. Gerando um total de $O(V^3 \log V)$.

7. Estudo do tempo de execução

Outra importante análise para determinar a funcionalidade de um programa, é a análise do tempo de execução. De nada adianta um aloritmo que nunca termina de rodar, ou que demora um tempo inacessível. “Um programa modesto que funciona é mais útil que um super-ambicioso que nunca funciona.”

Para essa análise em específico, os dados foram coletados experimentalmente. Fixou-se um número de vértices, variando o número de arestas, e o tempo de execução foi calculado. Num outro momento, variou-se o número de vértices, e o tempo de execução foi novamente calculado. Com os dados em mãos, foi possível perceber qual seria o comportamento do algoritmo para cada variável.

→ Análise de tempo seção 7 – Variação do número de vértices: Tabela 1.

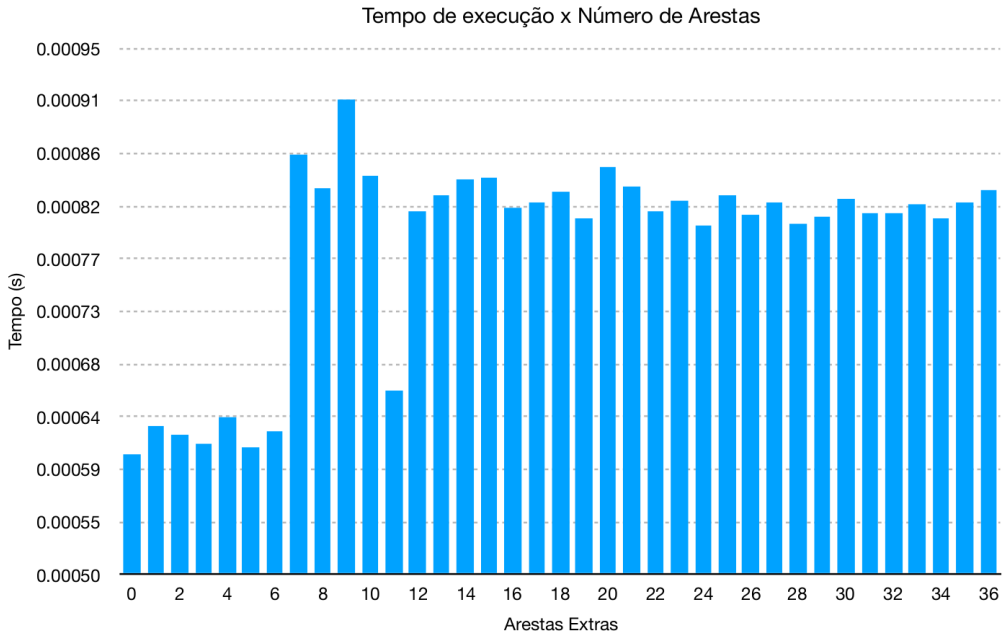
Grafo 1	**5 vértices**	**0.001464**	**MST única!**
Grafo 2	**50 vértices**	**0.005625**	**MST única!**
Grafo 3	**500 vértices**	**0.049263**	**MST não única!**
Grafo 4	**5000 vértices**	**11.227826**	**MST não única!**
Grafo 5	**5000 vértices (mais denso)**	**20.620275**	**MST não única!**

Importante notificar que as informações da tabela estão em segundos, e que não foi possível montar um gráfico para esta seção por termos uma diferença muito grande nos valores, e não conseguirmos encaixá-los em uma escala apropriada.

Analisando os dados acima, consegue-se perceber com facilidade o grande impacto do número de vértices no tempo de execução do programa, gerando uma enorme diferença para grafos muito grandes.

Portanto, pode-se confirmar a análise feita anteriormente, na qual o programa segue uma complexidade cúbica, visualmente expressa pelo tempo e pela densidade.

→ Gráfico seção 7 – Figura 6.



A partir dos dados presentes no gráfico é possível deduzir a dependência entre o tempo de execução do programa e o número de arestas, ou seja, a densidade do gráfico. Os dados foram obtidos ao executar-se o programa diversas vezes sobre um mesmo gráfico base com 10 vértices e o mínimo de conexões (ou seja, no valor 0, o gráfico é uma MST), adicionando a cada nova rodada de execução uma aresta extra com peso aleatório (entre 3, 5 ou 7) até que estivesse o mais denso possível e nenhuma outra conexão fosse possível entre os vértices.

Analisando, primeiramente, o intervalo de 0 a 6 arestas extras, é possível inferir que o número de arestas não interveio de forma expressiva no tempo de execução, que se mantém em torno de um mesmo valor médio. É importante dizer que durante esse intervalo há apenas uma única MST.

Em seguida, nota-se o intervalo 7 a 10, no qual observa-se um aumento de aproximadamente 40% no tempo de execução. Nesse momento encontra-se uma segunda MST. Esse incremento no tempo, portanto, é esperado já que a função algorítmo Primm é chamado duas vezes com o parâmetro que utiliza a função de inserção organizada na lista

de saída, que possui um grande tempo de execução em comparação com outras funções.

Analisando exclusivamente o intervalo, o tempo de execução é aproximadamente o mesmo, indicando pouca dependência entre tempo e arestas.

O decaimento do tempo de execução em 11 ocorre devido ao fato de ser encontrada uma nova MST única e o algorítmo Primm ser chamado apenas uma vez. O tempo, no entanto, volta a girar em torno de um mesmo valor logo em seguida, quando uma segunda MST é encontrada. Esse valor médio é, no entanto, um pouco menor que o anterior (intervalo 7 a 10) pois a função de procura por segunda MST se torna mais eficiente já que a MST é encontrada em um vértice mais próximo ao primeiro.

8. Testes executados

Para que se possa ter noção do funcionamento do algoritmo, foram realizados diversos casos de teste. Alguns deles foram incluídos numa pasta *testes_executados,* com as configurações dos grafos, e suas respectivas saídas foram incluídas em outra pasta - *respostas_geradas,* bem como um arquivo chamado *resultado_testes,* que contém as respostas geradas no terminal de cada grafo testado, identificando-os pela ordem.

Assim, com esses dados em mãos, podemos analisar o comportamento do algoritmo quando submetido a diferentes situações. Será exemplificado nessa seção como foi gerada a saída, e o modo em que foi organizada.

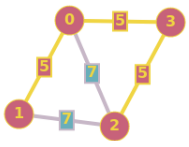
Daremos como exemplo três testes realizados, com suas respectivas saídas. As saídas esperadas seriam configurações das MST's destacadas em vermelho, em ordem crescente.

→ Desenho esquemático e Estrutura seção 8 – Figura 7: exemplo 1.

```
4
0 5 7 5
5 0 7 0
7 7 0 5
5 0 5 0
```

Entrada utilizada

Desenho do grafo e MST esperada



Saída gerada

0-1 / 0-3 / 2-3

MST com custo: 15

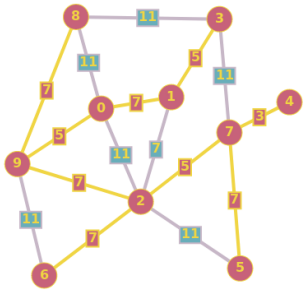
Economia vs malha totalmente conexa: 14
MST é única

→ Desenho esquemático e Estrutura seção 8 – Figura 8: exemplo 2.

Entrada utilizada

```
10
0   7   11  0   0   0   0   0   11  5
7   0   7   5   0   0   0   0   0   0
11  7   0   0   0   11  7   5   0   7
0   5   0   0   0   0   0   11  11  0
0   0   0   0   0   0   0   3   0   0
0   0   11  0   0   0   0   7   0   0
0   0   7   0   0   0   0   0   0   11
0   0   5   11  3   7   0   0   0   0
11  0   0   11  0   0   0   0   0   7
5   0   7   0   0   0   11  0   7   0
```

Desenho do grafo e MST esperada



Saída gerada

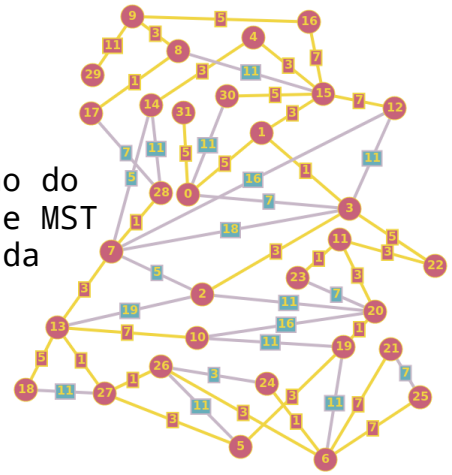
0-1 / 0-9 / 1-3 / 2-6 / 2-7 / 2-9 / 4-7 / 5-7 / 8-9
MST com custo: 53
Economia vs malha totalmente conexa: 73
MST não é única

→ Desenho esquemático e Estrutura seção 8 – Figura 9: exemplo 3.

Entrada utilizada

```
32
0 5 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 5
5 0 0 1 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 3 0 0 0 5 0 0 0 0 0 19 0 0 0 0 0 0 11 0 0 0 0 0 0 0 0 0 0 0 0
7 1 3 0 0 0 0 18 0 0 0 0 11 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 11 3 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 7 0 0 1 7 3 0 0 0 0 0 0
0 0 5 18 0 0 0 0 0 0 0 0 16 3 5 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 11 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 11 0 0
0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 11 16 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 11 0 0 0 16 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 19 0 0 0 0 3 0 0 7 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 3 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 0
0 3 0 0 3 0 0 0 0 11 0 0 0 7 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 0 0
0 0 0 0 0 3 11 0 0 0 11 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 11 0 0 0 0 0 0 16 3 0 0 0 0 0 0 0 1 0 0 0 7 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0
0 0 0 5 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0
0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 11 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 1 0 0 0 0 0
0 0 0 0 0 3 0 0 0 0 0 0 0 0 1 0 0 0 0 11 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 11 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Desenho do grafo e MST esperada



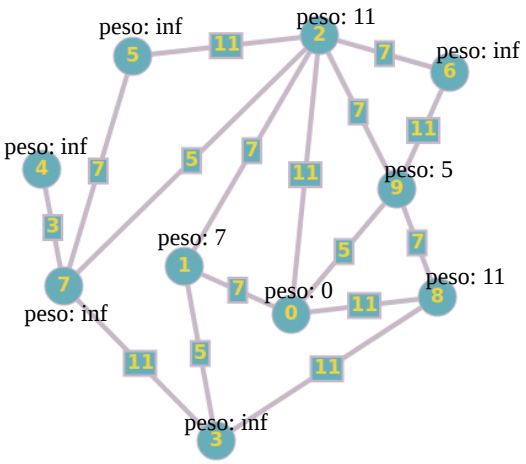
Saída gerada

0-1 / 0-31 / 1-3 / 1-15 / 2-3 / 3-22 / 4-14 / 4-15 / 5-19 / 5-27 / 6-21 / 6-24 / 6-25 / 6-26 / 7-13 / 7-28 / 8-9 / 8-17 / 9-16 / 9-29 / 10-13 / 11-20 / 11-22 / 11-23 / 12-15 / 13-18 / 13-27 / 15-16 / 15-30 / 19-20 / 26-27
MST com custo: 117
Economia vs malha totalmente conexa: 209
MST não é única

Agora, explicaremos passo a passo como o algortimo rodou no exemplo 2. Primeiro, atualizou-se a chave de todos os vértices para infinito, exceto a do vértice zero; que foi atualizada para zero, e inicluiu o vértice zero na MST – árvore binária.

Em seguida, checou-se para cada conexão do vértice zero se a chave do vértice de destino era maior que o peso da aresta; se sim, atualizou-se o peso. Chegou-se à seguinte configuração:

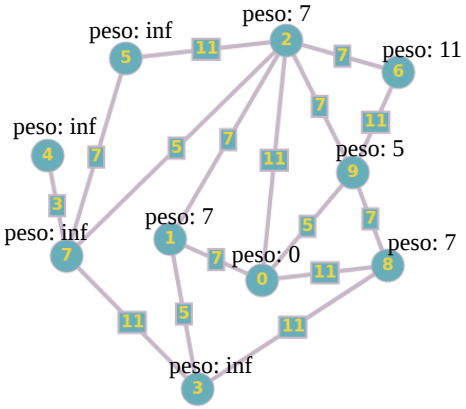
→ Desenho esquemático e Estrutura seção 8 – Figura 10: exemplo 2 - processo.



Então, escolheu-se o vértice de menor peso, a partir do zero, e repetiu-se o processo a partir do novo vértice. Nesse caso, escolheu-se o vértice nove, e inseriu-se o vértice nove na árvore binária de MST, assim como inseriu-se a aresta encontrada (a de menor peso) na lista de arestas, que será utilizada para printar no arquivo de saída.

Para repetir o processo, foi preciso verificar, para cada conexão do vértice nove se o vértice de destino já estava na MST. Caso contrário, verifica-se se a chave do vértice de destino é maior que o peso da conexão, se sim, atualiza-se a chave… Ao final do segundo processo, tem-se a seguinte configuração:

→ Desenho esquemático e Estrutura seção 8 – Figura 11: exemplo 2 – processo.



E assim, o algoritmo foi sendo aplicado, até que todos os vértices estejam na MST, e a lista de arestas tenha um total de V-1 arestas. Então,

organizou-se a lista de arestas para que fique em ordem crescente – tanto de cima para baixo, como da esquerda para direita.

Para encontrar a segunda MST, realizou-se o mesmo algoritmo repetidas vezes – para cada vértice do grafo, comparando-se sempre a lista de arestas da primeira MST com a lista de arestas da nova MST gerada. Caso para algum vértice a MST gerada seja diferente da primeira, printou-se no terminal que a MST não é única e gerou-se um segundo arquivo contendo a nova MST encontrada.

Nesse caso pode-se obter a MST como já demonstrada anteriormente, e pode-se obter também a segunda MST trocando a aresta 0-1 de peso sete, pela aresta 1-2 também de peso sete.