Executar pela linha de comando

O arquivo que contém a main é o **App.java**

Se tentarmos compilar com *javac App.java*, receberemos o seguinte erro:

```
pedro in Documents/PedroMoraes/src
-> javac App.java
./Vertex.java:5: error: package org.apache.commons.math3.util does not exist
import org.apache.commons.math3.util.Pair;
                                    ^
```

Para compilar, precisamos passar um parâmetro na linha de comando.

Para compilar e executar sem erros, é só copiar as linhas abaixo

javac -cp ".:commons-math3-3.6.1.jar" App.java

java -cp ".:commons-math3-3.6.1.jar" App.java grafo1.txt

Para compilar no Windows, substitua o ':' de dentro das aspas por ';'

```
pedro in Documents/PedroMoraes/src
-> javac -cp ".:commons-math3-3.6.1.jar" App.java
pedro in Documents/PedroMoraes/src took 3s
-> java -cp ".:commons-math3-3.6.1.jar" App g1.txt
Grafo original
1 --- 5 --- 2
1 --- 5 --- 3
1 --- 4 --- 4
2 --- 4 --- 3
3 --- 4 --- 5
4 --- 7 --- 7
4 --- 7 --- 6
5 --- 6 --- 7
5 --- 6 --- 8
6 --- 6 --- 7
6 --- 6 --- 9
7 --- 5 --- 8
7 --- 6 --- 9
8 --- 8 --- 9

----------------

MST
1 --- 5 --- 2
1 --- 4 --- 4
2 --- 4 --- 3
3 --- 4 --- 5
5 --- 6 --- 7
6 --- 6 --- 7
6 --- 6 --- 9
7 --- 5 --- 8
```

Introdução

O algoritmo de prim é um algoritmo guloso que, dado um grafo que seja conexo, valorado e não direcionado, encontra uma árvore geradora de custo mínimo.

Por ser um algoritmo guloso, o algoritmo de prim busca uma solução comum para problemas de otimização da seguinte forma:

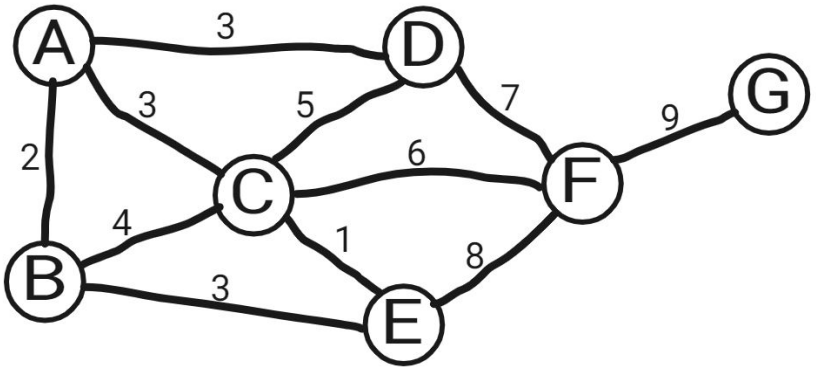
- Realiza a escolha que parece ser a melhor no momento na esperança de que a mesma acarrete em uma solução ou prevenção de futuros problemas a nível global
- É simples e de fácil implementação
- É míope: ele toma decisões com base nas informações disponíveis na iteração corrente

Exemplo

O passo a passo é relativamente simples:

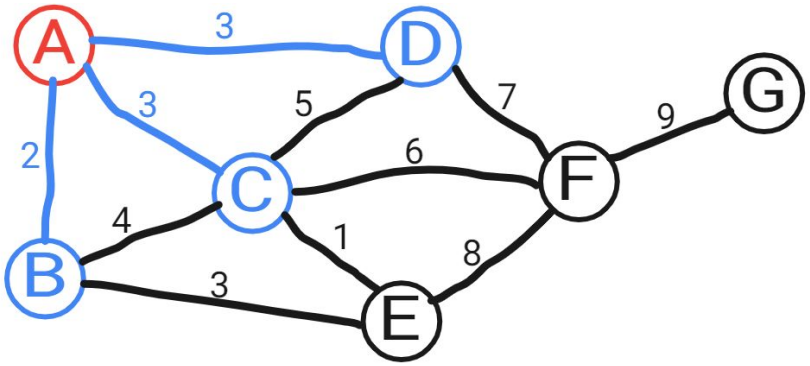
1. Escolhe um vértice
2. Guarda esse vértice numa lista. Vou chamar essa lista de *"Visitados"*
3. A partir desse vértice, escolhe a aresta de menor custo conectada a um vértice que não esteja presente na lista *Visitados*
4. Adiciona o vértice ligado a essa aresta a lista *Visitados*
5. Escolhe um dos vértices da lista *Visitados* e repete os passos 3, 4 e 5 até que todos os vértices do grafo estejam na lista *Visitados*

Vou exemplificar a execução do algoritmo de prim no seguinte grafo:



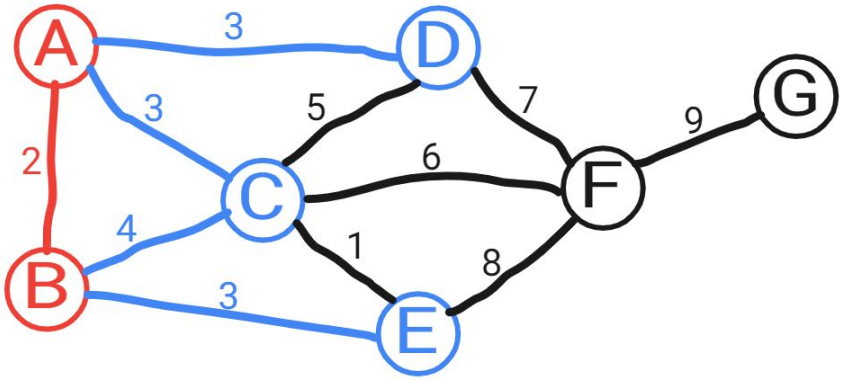
Visitados = { }

Para a execução do algoritmo, precisamos escolher um vértice qualquer como ponto de partida e a partir dele executar o algoritmo. Vamos "guardar" quais vértices já visitamos na lista *Visitados*. Nesse exemplo, vou começar pelo vértice "A".



Visitados = { A }

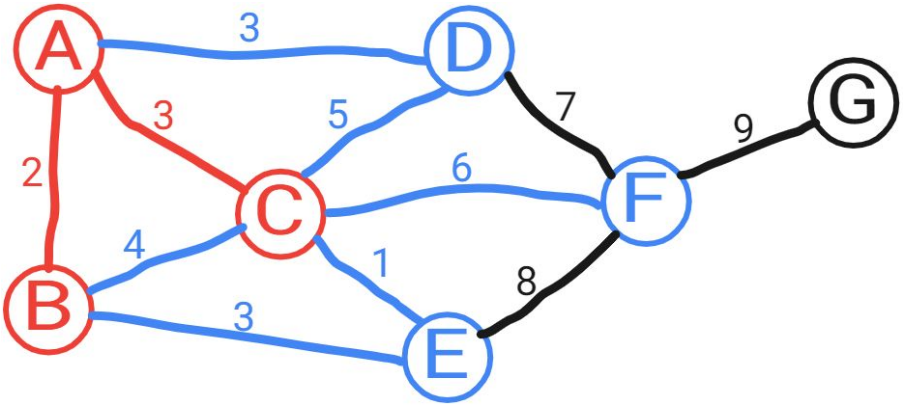
Agora, precisamos escolher a aresta com menor custo conectada a um vértice que não esteja na lista *Visitados*. Nesse caso, a aresta que liga A e B é a de menor custo e B não foi visitado.



Visitados = { A, B }

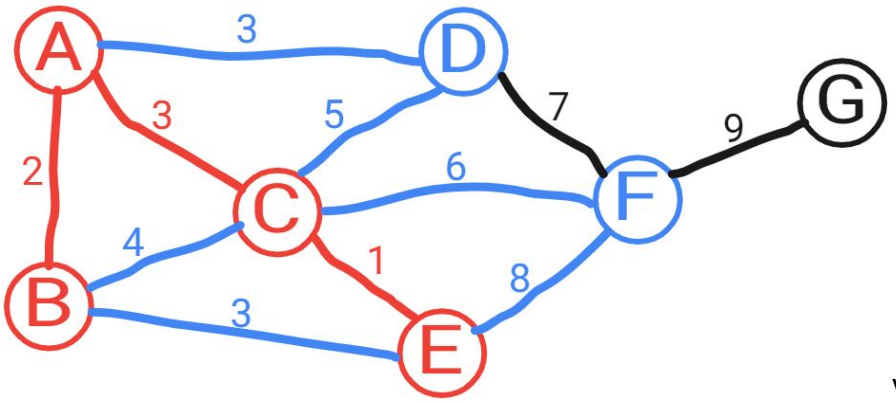
Os vértices e a aresta pintados de vermelho fazem parte da árvore geradora mínima que estará completa ao fim do algoritmo.

Agora, olhamos para todos os os vértices que podem ser alcançados a partir de A e B (são os vértices presentes na lista *Visitados*) e realizamos o mesmo processo anterior de escolher a aresta de menor custo conectada a um vértice não visitado. As arestas que ligam A-D, A-C e B-E possuem valor 3, nesse caso, posso escolher qualquer uma das 3. Vou escolher a que liga A-C.



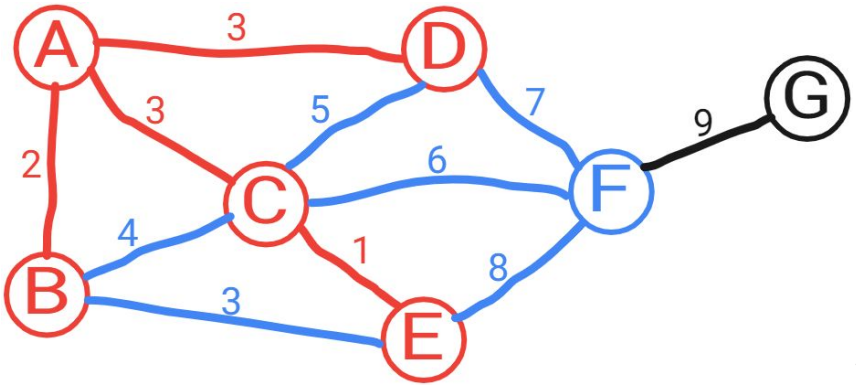
Visitados = { A, B, C }

Agora, repetindo o mesmo passo a passo, a aresta de menor custo é a que liga os vértices C e E, então:



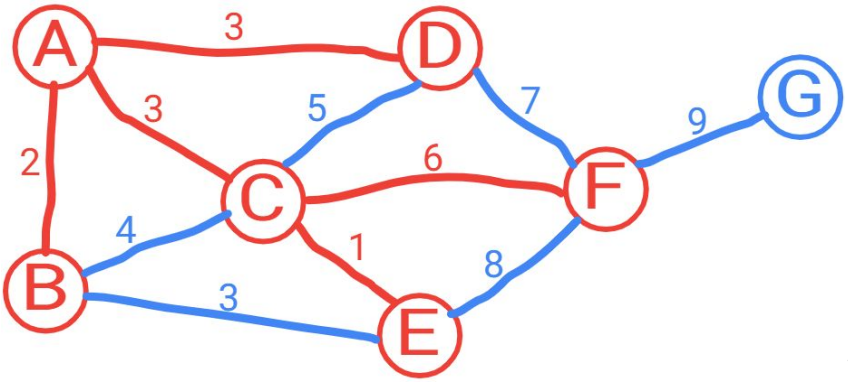
Visitados = {A, B, C, E}

Devemos escolher agora a aresta que liga os vértices A e D pois é a de menor custo.



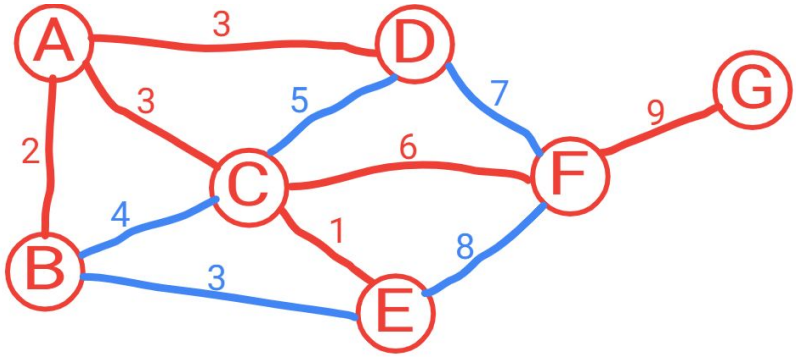
Visitados = {A, B, C, E, D}

Embora a aresta que liga os vértices B e E seja a de menor valor, ambos os vértices já foram visitados, então ela não será selecionada. Lembrando que é preciso selecionar uma aresta que conecta um vértice visitado a um não visitado. A de menor valor é a que conecta C e F.



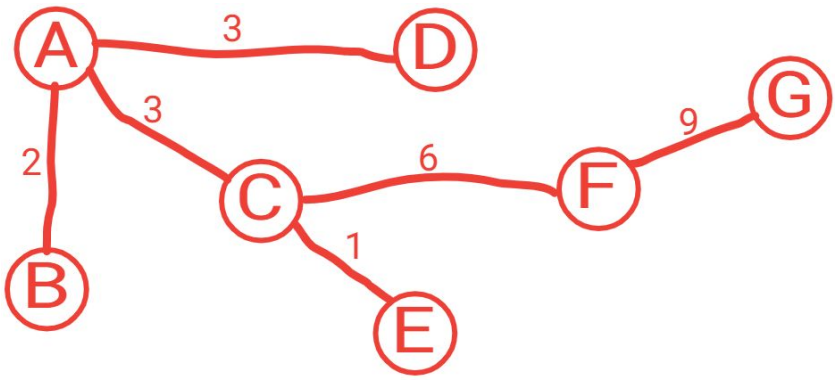
Visitados = {A,B,C,E,D,F}

Agora só falta incluir o vértice G, como só há uma aresta conectada a ele, ela será a escolhida:



Visitados = {A,B,C,E,D,F,G}

Agora, todos os vértices estão conectados, formando uma árvore.



Implementação

Para implementação do algoritmo, foram criadas 4 classes:

- App: Classe contendo a main. Realiza a leitura do arquivo contendo o grafo e inicializa a execução do algoritmo de Prim.
- Prim: Classe contendo o algoritmo de Prim e funções para printar o grafo e a árvore geradora.
- Edge: Classe das arestas
- Vertex: Classe dos vértices

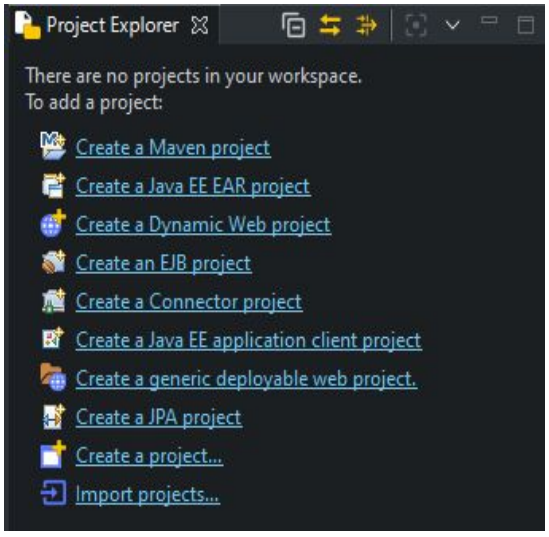
A estrutura do algoritmo pode ser resumida no print a seguir. O método *execute* roda o Algoritmo de Prim a partir de uma lista de vértices chamada *graph*. Se o grafo for conexo, o algoritmo vai rodar naturalmente. Caso seja desconexo, a execução será finalizada.
A partir de um grafo conexo, a execução segue normalmente. O algoritmo de Prim nos permite começar de qualquer um dos vértices do grafo. No meu programa, decidi começar sempre do primeiro vértice da lista apenas por facilidade. Então, a partir do primeiro vértice, ele itera por todas as arestas procurando a de menor custo. Ao encontrar, o vértice ligado a ela é marcado como visitado e ela volta a procurar a aresta de menor custo de um dos vértices já visitados. Funciona exatamente como demonstrei neste relatório.

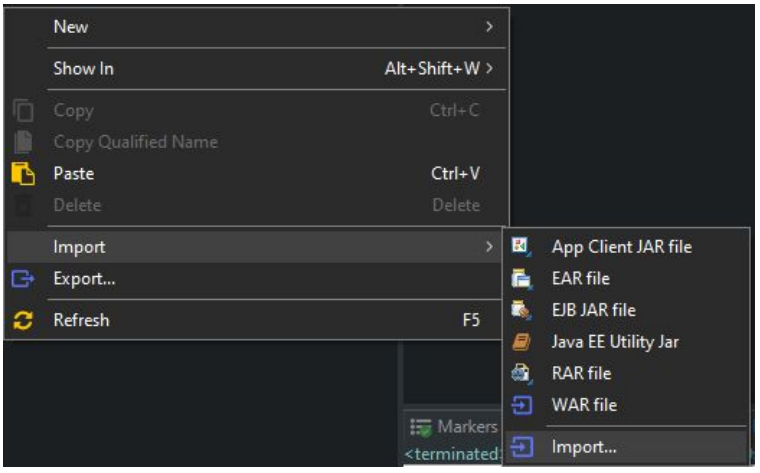
```java
private List<Vertex> graph;

public void execute(){
    if (graph.size() > 0){
        graph.get(0).setVisited(true);

        while (isConnected()) {
            Edge nextMinimum = new Edge(Integer.MAX_VALUE);
            Vertex nextVertex = graph.get(0);
            for (Vertex vertex : graph){
                if (vertex.isVisited()){
                    Pair<Vertex, Edge> candidate = vertex.nextMinimum();
                    if (candidate.getValue().getPeso() < nextMinimum.getPeso()){
                        nextMinimum = candidate.getValue();
                        nextVertex = candidate.getKey();
                    }
                }
            }
            nextMinimum.setIncluido(true);
            nextVertex.setVisited(true);
        }
    } else {
        System.out.println("Não é um grafo. Parando execução do programa...");
        System.exit(-1);
    }

}
```

Execução

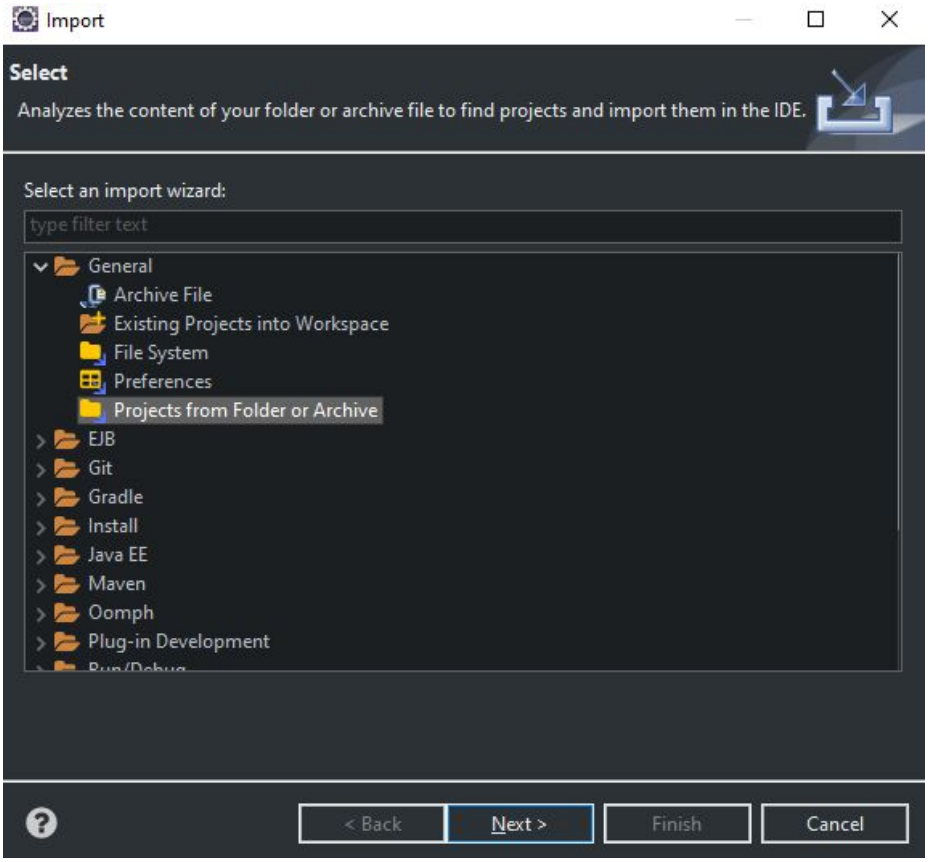
Para executar, recomendo que importe o projeto na IDE de sua preferência e rode a classe App passando como argumento o path do arquivo contendo o grafo. Vou exemplificar esse passo a passo no Eclipse, a IDE que utilizei para desenvolver o trabalho.

1. Na aba Project Explorer, clicar em *Import projects*, ou então *botão direito do mouse* > *Import* > *Import*

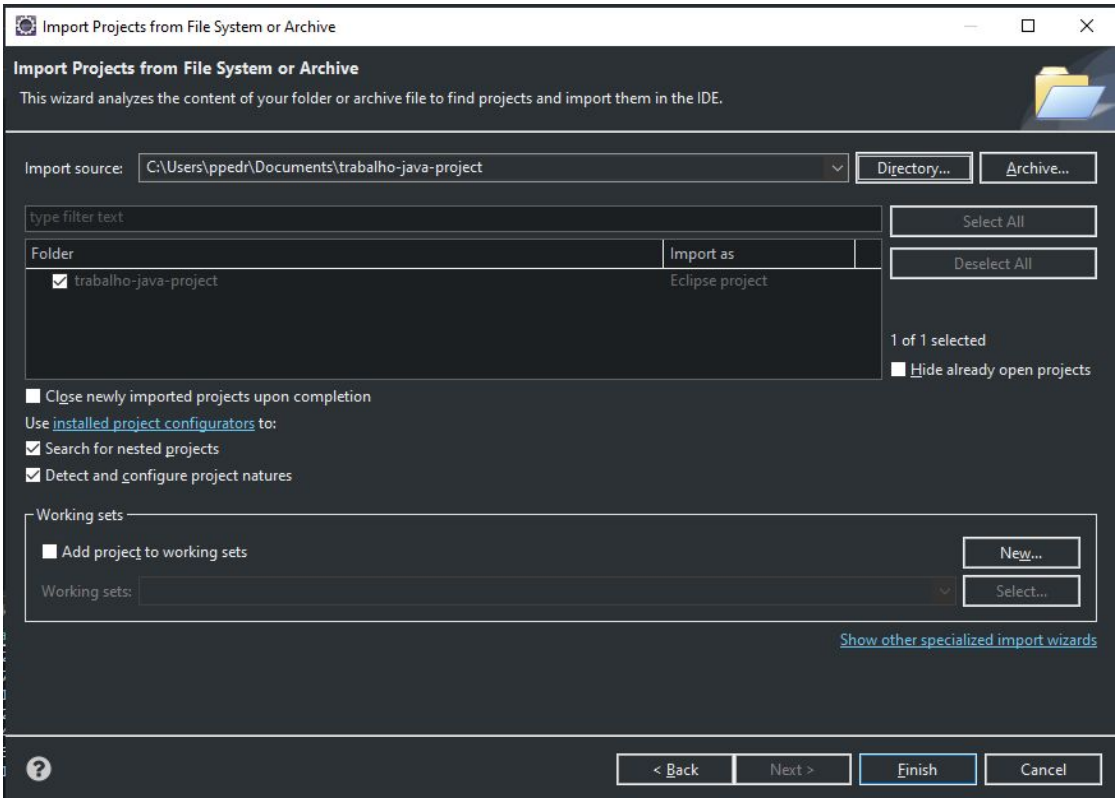




2. A partir da janela que abriu, na aba *General*, escolher a opção *Projects from Folder or Archive*

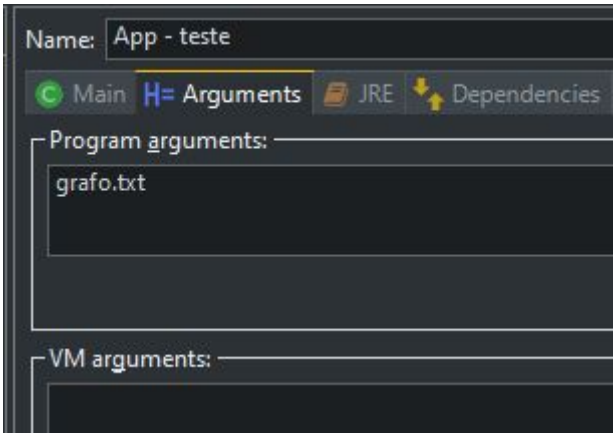


3. Então, selecione a pasta do projeto e clique em *Finish*



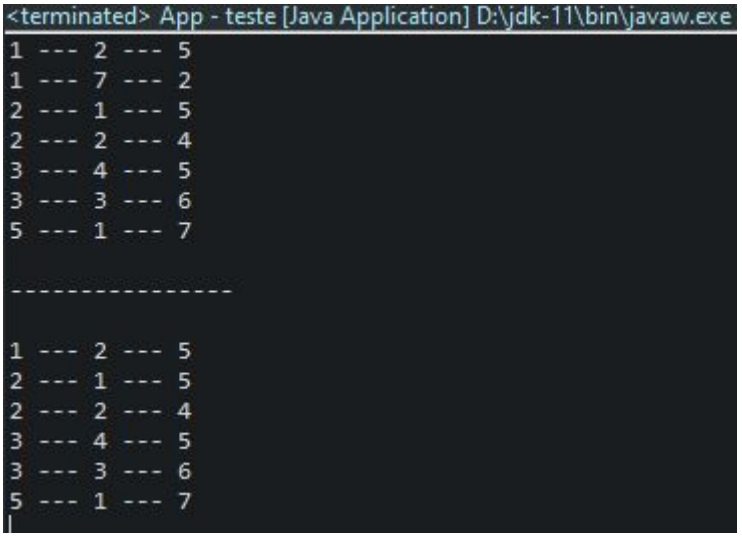
4. Agora, clique com o botão direito na pasta do projeto, *Run as* > *Run configuration*

5. Na janela que abriu, selecione a aba *Arguments* e adicione o path do arquivo contendo o grafo. No caso desse projeto, estou incluindo na raiz do projeto o arquivo grafo.txt como argumento.



6. Então, agora é só rodar o programa pelo botão de run na barra superior de ícones ou clicando com *botão direito no projeto* > *Run As* > *Java Application*

Abaixo um print de uma execução



É possível executar o programa pela linha de comando, no entanto, algumas versões do JAVA retornam erro ao compilar pois não encontram a biblioteca utilizada na implementação. A fim de evitar maiores problemas, estou incluindo uma pasta contendo a biblioteca utilizada e classes já compiladas sem erro. Essas classes estão na pasta *bin*.

Para executar pelo terminal, basta executar a classe App contida dentro da pasta *bin*. Se quiser compilar, basta compilar o arquivo App.java e executá-lo passando o path do arquivo que contém o grafo. Primeiro o Grafo original será printado e então sua árvore geradora mínima.