Projeto de Rede de Distribuição de Água

Jonas de Oliveira Freire Filho

Instituto Metrópole Digital - IMD / Universidade Federal do Rio Grande do Norte - UFRN

Resumo

Este presente trabalho tem como objetivo propor uma modelagem em grafos que resolva a seguinte problemática: Dados n clientes e suas localizações, calcule a menor quantidade de tubulações em metros para um projeto de distribuição de água. Para isso, vai-se problematizar o tema e discorrer sobre a proposta de solução que usufruirá do conceito de árvore geradora mínima.

Palavras-chave: modelagem, grafos, distribuição de água, árvore geradora mínima

1. Introdução

Para desenvolver um projeto de rede de distribuição de água, precisa-se encontrar uma forma para representar n clientes, um reservatório de água e suas devidas localizações, assim como, as tubulações para modelar esse projeto. Ademais, também necessita-se que esse projeto seja intuitivo e eficiente, isto é, que economize o máximo de recursos.

Para representar e propor uma forma eficiente de desenvolver o projeto, vai-se usufruir do conceito de grafos e de árvore geradora mínima.

2. Descrição do problema real

Uma instituição responsável pela distribuição de recursos hídricos de uma cidade precisa criar um projeto para partilhar água potável pelo município, de

*Autor correspondente

Email address: jonas.oliveira.111@ufrn.edu.br (Jonas de Oliveira Freire Filho)

modo a suprir a necessidade hidráulica de todos os seus clientes e economizar o máximo de recursos possíveis. Para este fim, a instituição usufruirá da seguinte estratégia: calcular a combinação de tubulações que passe por todos os seus clientes e que, ao total, use a menor quantidade, em metros, de tubos.

Para alcançar o objetivo, pretende-se modelar o projeto com grafos, de forma que a partir da localização de seus clientes, consiga gerar a combinação almejada.

3. Modelagem em grafos

Dado o problema descrito, este tópico propõe uma modelagem em grafos não direcionado para resolvê-lo.

A representação em grafos ocorrerá da seguinte forma: os vértices retratarão os clientes e um único reservatório do sistema, que chamaremos de estabelecimentos, juntamente com suas localizações; as arestas vão simbolizar a tubulação que vai interligar dois estabelecimentos, sendo seu peso a quantidade mínima, em metros, de tubos necessária para tal interligação. E, desse modo, tem-se uma uma forma satisfatória para representar computacionalmente o problema, como ilustra a Figura 1.

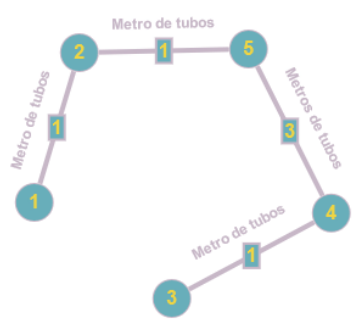

Figura 1: Representação do problema em grafos.

Outrossim, para chegar nesse grafo que representa a solução, deve-se criar o grafo sem arestas, contendo somente os vértices, ou seja, os estabelecimentos e suas localizações, como mostra a Figura 2.

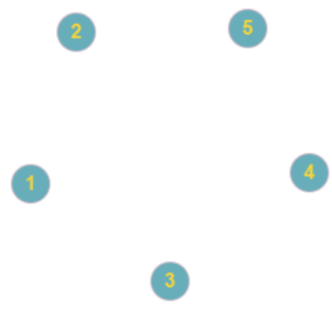

Figura 2: Grafo sem arestas.

Em consequência desse grafo e das distancias entre seus estabelecimentos, que pode ser calculada pela triangulação de suas localizações, pode-se gerar o grafo completo que possua nos pesos de suas arestas, a quantidade de metros de tubos necessária para interligar cada par de estabelecimento, como mostra a Figura 3.

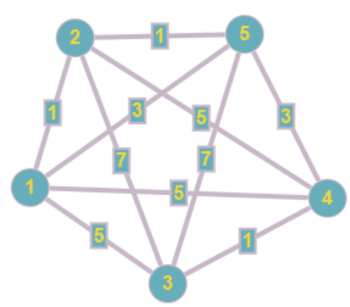

Figura 3: Grafo completo.

Com o grafo completo, pode-se gerar a árvore geradora mínima, que, pela sua definição, será a árvore geradora do grafo que possui o custo mínimo, isto é, o grafo com a combinação de tubulações que passa por todos os seus estabelecimentos, em que a soma de todos os pesos de suas arestas (metros de tubos) é a menor possível, que, não coincidentemente, é o que esperava-se obter, ver Figura 4.

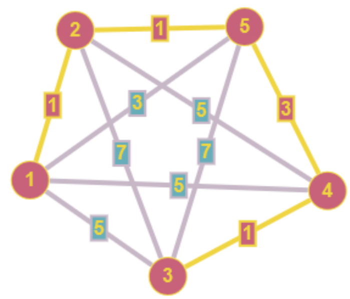

Figura 4: Grafo completo, com árvore geradora mínima em vermelho.

4. Estado da Arte

Na literatura de grafos, o problema de encontrar o caminho de custo mínimo para um determinado grafo conexo, se caracteriza como a árvore geradora mínima e já foi largamente discutida e aprimorada com o decorrer do tempo. Neste tópico, irar-se-á introduzir os principais algoritmos e autores que contribuíram para esta modelagem, além de explicar como os testes desse trabalho vão ser implementados.

O primeiro algoritmo a ser apresentado é o de Joseph Bernard Kruskal Jr., que, em 1956, publicou no jornal *Proceedings of the American Mathematical Society* seu artigo *On the shortest spanning subtree of a graph and the traveling salesman problem* [1] propondo um dos principais algoritmos para gerar a árvore geradora mínima de um grafo conexo ponderado não direcionado G. Ele sugeriu fazer o seguinte procedimento:

1. Criar uma floresta em que cada vértice do grafo G será uma árvore dela.
2. Criar um conjunto de arestas contendo todas as arestas do grafo G.
3. Enquanto existir arestas dentro do conjunto e a floresta não for conexa o algoritmo fará:

(a) Removerá a aresta de menor peso do conjunto.

(b) Adiciona essa aresta na floresta, de modo a combinar duas arvores em uma só, caso essa aresta não forme ciclo.

E dessa forma, o algoritmo gera a árvore geradora mínima para G, com complexidade $O(n \log n)$, para n igual ao número de arestas.

Segue o pseudocódigo do algoritmo:

```
algorithm Kruskal(graph) is
    forest := ∅

    for each vertice in graph.vertices do
        MAKE-SET(vertice)

    ordenedEdges := graph.edges ordened by weight
    for each (vertice1, vertice2) in ordenedEdges do
        if FIND-SET(vertice1) ≠ FIND-SET(vertice2) then
            forest := forest ∪ {(vertice1, vertice2)}
            UNION(FIND-SET(vertice1), FIND-SET(vertice2))

    return forest
```

Figura 5: Pseudocódigo do algoritmo de Kruskal.

O segundo algoritmo a ser introduzido foi criado por Robert Clay Prim, em 1957, em seu artigo: *Shortest connection networks and some generalizations* [2] publicado no jornal *Bell System Technical Journal*. Ele propôs uma abordagem diferente da de Kruskal para gerar a árvore geradora mínima de um grafo ponderado completo não direcionado G. Segue o algoritmo:

1. Inicializa uma árvore com um vértice arbitrário do grafo G.
2. Adiciona a árvore a aresta de menor peso que conecta um vértice que já esta na árvore com outro vértice que não está na árvore.
3. Repete o segundo passo até que todos os vértices estejam na árvore.

Desse modo, Prim gera a árvore geradora mínima para G, com complexidade $O(n \log m)$, para n igual ao número de arestas e m igual ao número de vértices.

Segue o pseudocódigo do algoritmo:

```
algorithm Prim(graph) is
    randomVertice := graph.vertices.random

    setOfVerticesIncluded := randomVertice
    setOfVerticesNotIncluded := graph.vertices - randomVertice

    setOfEdges := ∅

    while setOfVerticesIncluded ≠ graph.vertices do
      newEdge := FIND-NEW-EDGE() // Find edge of minimum weigth between
                                 // vertices included and not included

      setOfEdges :=  setOfEdges U newEdge
      setOfVerticesIncluded := setOfVerticesIncluded U newEdge.verticeNotIncluded
      setOfVerticesNotIncluded := setOfVerticesNotIncluded - newEdge.verticeNotIncluded

    return NEW-GRAPH(setOfVerticesIncluded, setOfEdges)
```

Figura 6: Pseudocódigo do algoritmo de Prim.

Após implementação do projeto, o autor pretende testar a aplicação desenvolvida realizando análises empíricas. Para isso, vértices (estabelecimentos) serão gerados aleatoriamente, bem como suas coordenadas (localização) e a partir delas irá-se-á gerar o grafo completo, para finalmente executar o algoritmo que encontra a árvore geradora mínima, gerando, assim, a solução do problema.

Toda a implementação do projeto será desenvolvida pelo autor deste artigo, incluindo os casos de teste, com ele podendo disponibilizar os dados gerados a fito de verificar a corretude do algoritmo. O número de vértices a ser gerados e a quantidade de testes que irá ser realizada não será especificada agora, pois vai depender do ambiente que os testes vão ser executados.

Ademais, tem-se um *link* relevante para este trabalho, graphonline.ru [3] que foi usado para gerar reapresentações gráficas de alguns grafos mostrado nesse artigo.

5. Proposta de Abordagem Algorítmica

O algoritmo escolhido pelo autor para implementação deste trabalho foi o de Prim, ver Figura 6, pois, além de ser eficiente, possui uma abordagem simplória nos quesitos de entendimento e de implementação.

6. Descrição do Algoritmo Implementado

Este tópico objetiva explicar detalhadamente o algoritmo desenvolvido pelo autor. Mostrando, assim, as classes criadas, suas estruturas, relações e principais funções, incluindo, a de achar a arvore geradora mínima.

O trabalho foi implementando usando a linguagem *Java* usando a ferramenta *Maven* e a biblioteca *JUnit* e todos o projeto esta armazenado neste repositório. No total, foram implementadas seis classes, são elas: *Coordinates*, *Node*, *Graph*. E as classes para teste: *GenerateGraphs*, *ObjectGraphForAnalysis* e *GraphTests*.

Primeiramente, tem-se a classe *Coordinates*, que é responsável por armazenar as coordenadas dos clientes da empresa que irá fornecer a rede de distribuição de água. Nela são armazenadas a latitude e longitude da coordenada.

A segunda classe, *Node*, representa os clientes da empresa. Nela são armazenadas o nome do cliente e sua coordenada. Essa é uma das classes principais do projeto. E nela foi implementada um relevante método, *calculeDistance*, que calcula a distancia de um cliente para o outro.

A terceira classe, *Graph*, é responsável por armazenar os clientes da empresa. Essa classe, é, de fato, a classe principal do projeto. São os métodos dela que permitiram a geração grafo completo e a árvore geradora mínima.

Ademais, tem-se as classe que foram usadas para fazer os testes. A *GenerateGraphs* é uma classe abstrata e foi usada para gerar grafos automaticamente, desde adicionar os nós ou cliente ao grafo até gerar o grafo completo a partir de uma combinação de *seed* e do tamanho do grafo.

A classe *ObjectGraphForAnalysis*, é responsável por armazenar as informações durante os testes. Ela armazena o grafo completo, a arvore geradora mínima, a *seed* do grafo e o tempo em nanossegundos necessário para gerar a arvore geradora mínima.

E, por ultimo, tem-se a classe *GraphTests* que é responsável por executar os testes do projeto. Essa classe vai ser abordada em mais detalhes na próxima seção.

7. Descrição dos Experimentos Computacionais

Neste tópico o autor irá explica como os testes foram implementados e como reproduzir os testes. Além de especificar em que ambiente os testes do autor foram executados.

Os testes, como já foi explicado no capítulo anterior, foram implementados na classe $GraphTests$ com a biblioteca $junit5$. E para execução dos testes, é declarado os valores de 5 variáveis:

- $SPREAD$, indica a diferença quantidade de clientes de um teste para o outro;
- $INITIAL\_AMOUNT\_OF\_NODES$, que indica a quantidade inicial de clientes que vai ser executada no primeiro teste;
- $AMOUNT\_OF\_TESTS$ que indica a quantidade de teste que vai ser feito;
- $ITERATIONS$ que indica a quantidade de vezes que cada teste irá ser realizado, note que nesse caso a média será calculada e somente ela será levada em conta para as análises; e
- $SEED$ que armazena a *seed* (semente) que vai ser usada para gerar todos os dados a serem inseridos em todos os grafos.

Os testes consistem em adicionar uma quantidade predefinida de clientes em localizações aleatórias, geradas pela *seed* e então gerar o grafo completo, calculando todas as distâncias possíveis de todos os clientes para todos os clientes e então gerar a árvore geradora mínima desse grafo e mensurar o seu tempo. E realizar isso várias vezes e para diversas quantidade de clientes.

As declarações das variáveis ocorrem no método $analyisManyShortestSpanningSubtree$ que é o método que irá executar os testes, como mostra a Figura 7.

```java
@Test
public void analyisManyShortestSpanningSubtree() {
    int SPREAD = 10;
    int INITIAL_AMOUNT_OF_NODES = 100;
    int AMOUNT_OF_TESTS = 5;
    int ITERATIONS = 50;
    long SEED = 0;

    printInfo(SPREAD, INITIAL_AMOUNT_OF_NODES, AMOUNT_OF_TESTS, ITERATIONS, SEED);

    List<Long> seeds = GenerateGraphs.generateSeeds(ITERATIONS, SEED);

    for (int i = 0; i < ITERATIONS; i ++) {
        analysisMeanOfOneShortestSpanningSubtree(
                INITIAL_AMOUNT_OF_NODES + (i*SPREAD),
                AMOUNT_OF_TESTS,
                seeds.get(i)
        );
    }
}
```

Figura 7: Método *analyisManyShortestSpanningSubtree*.

Perceba que, nessa figura é possível ver os valores das variáveis que foram usadas para os testes realizados pelo autor. Os testes foram rodados em uma máquina com processador *11th Gen Intel(R) Core(TM) i7-11390H @ 3.40GHz 2.92 GHz* com memória *16,0 GB (utilizável: 15,7 GB)* e sistema operacional *Windows 11 Home Single Language - v21H2*.

Para reproduzir os testes, pode-se usar qualquer IDE Java, no caso do autor, ele usou o Intellij IDEA da JebBrains e fez o seguintes passos:

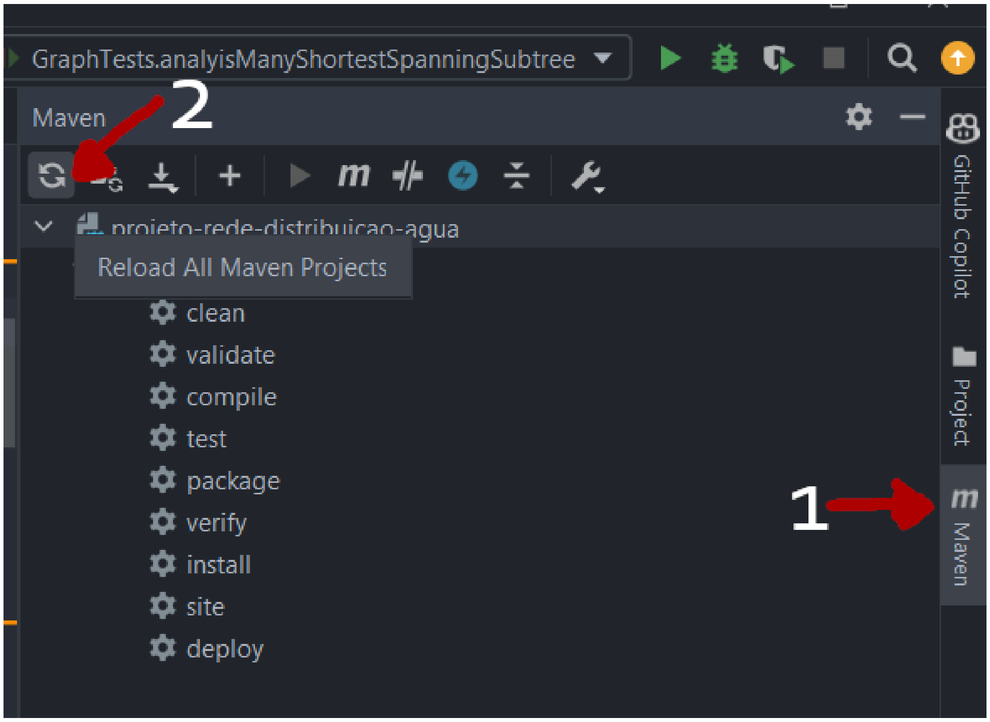

Figura 8: Passo um.

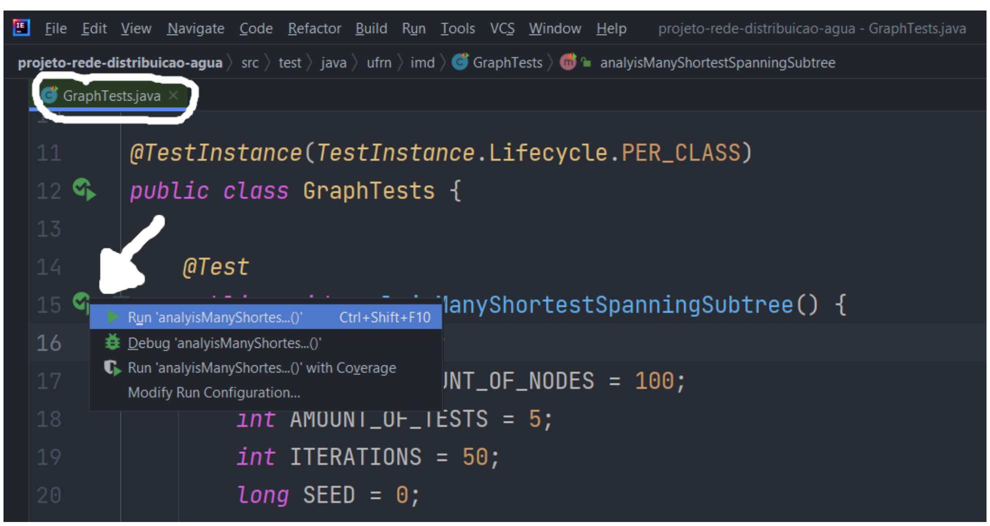

Figura 9: Passo dois.

No passo um: Abrir o projeto e na aba Maven que fica a direita, clicar em *Reload All Maven Projects*. No passo dois: Abrir a classe *GraphTests* e na linha 15 clicar no simbolo verde e em seguida clicar *Run 'analyisManyShortes...()'*. E, desse modo, os testes vão ser executados e apareceram todas as informações e dados no terminal, como mostra a Figura 10.

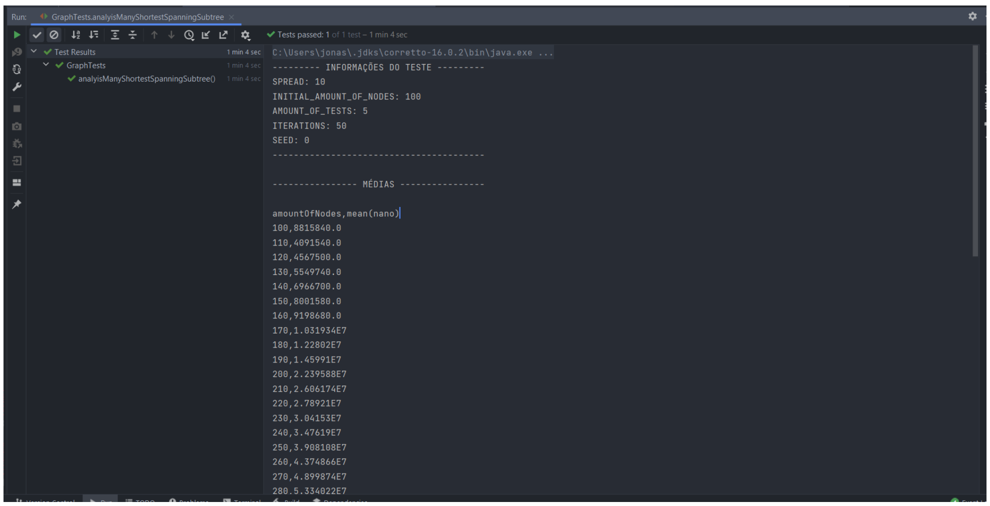

Figura 10: *Output* do teste no terminal.

8. Resultados Obtidos

Os teste que o autor fez foram salvos no arquivo *result.csv* e pode ser encontrado no repositório deste trabalho. Segue o gráfico com os resultados encontrados.

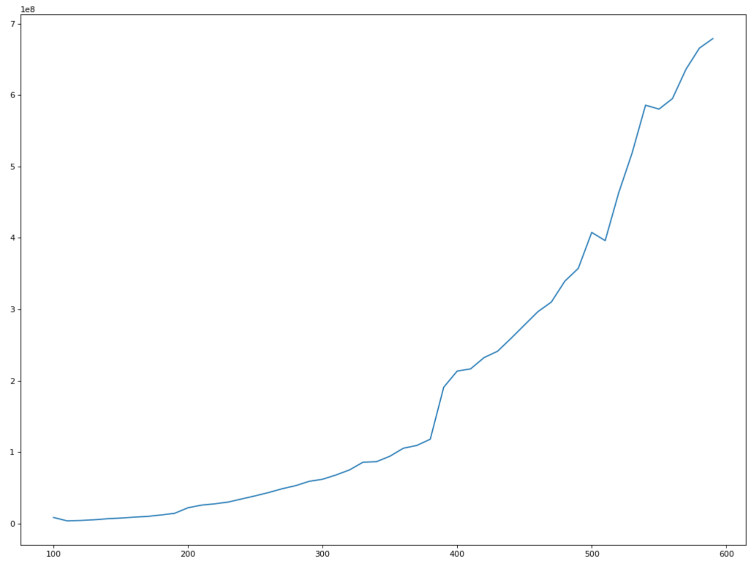

Figura 11: Gráfico com os resultados.

Com este gráfico é possível perceber que ele realmente caracteriza uma complexidade $O(n \log(n))$ como esperado. Note que o gráfico possui alguns picos e baixos, mas isso ocorre devido a instabilizações do ambiente em que os testes foram rodados.

9. Conclusão

Desse modo, espera-se que este presente trabalho tenha discorrido sobre o cenário a ser modelado por grafos, tenha definido claramente o modelo proposto, bem como, tenha explicado satisfatoriamente como este modelo implicará na solução do problema para n clientes.

E que tenha ficado claro todo o desenvolvimento deste trabalho, incluindo sua códificação os seus testes e seus resultados.

Referências

[1] J. B. K. Jr., On the shortest spanning subtree of a graph and the traveling salesman problem, Proceedings of the American Mathematical Society 7 (1)

(1956) 48–50. doi:10.1090/S0002-9939-1956-0078686-7.

[2] R. C. Prim, Shortest connection networks and some generalizations, Bell System Technical Journal 36 (6) (1957) 1389–1401. doi:10.1002/j.1538-7305.1957.tb01515.x.

[3] O. Shiakhatarov, Crie grafos e encontre o caminho mais curto ou use outro algoritmo, https://graphonline.ru/pt/. Acessado em: 11 de maio de 2022.