**Uso de grafos para melhor escolha de rota em empresa de frete**

**ALUNOS**: Rogério Lacerda dos santos, Gabriel Melo de Menezes

**INSTITUIÇÃO:** Universidade Federal Rural do Agreste de Pernambuco (UFAPE)

**DISCIPLINA:** AED II

Grafos são comumente utilizados na matemática para representar um conjunto de vértices, que podem possuir ou não uma correlação entre os vértices. Tal correlação é chamada de arestas.

O objetivo desde relatório é apresentar uma solução utilizando grafos que apresente o menor caminho entre duas cidades. Iremos tratar os locais como os vértices. Já a estrada que liga os dois locais será tratada como a aresta. Cada aresta terá um peso, que representará a distância em KM.

**ÍNDICE**

1. Introdução ao problema
2. Grafo escolhido e sua implementação
3. Desenho das arestas e vértices
4. Conclusão
5. Introdução

**- Empresa de frete**

O problema proposto a ser solucionado por meio do uso de grafos é um cenário onde uma empresa de frete precise decidir qual a melhor rota para realizar sua entrega. Como sabemos um destino normalmente possui mais de um caminho, tais caminhos nós pensaremos como as arestas de um grafo. O programa terá o trabalho de calcular todos os possíveis caminhos até o destino, e apontar aquele que será o menor caminho. Para fazer esta conta, nós usaremos as distâncias entre as cidades que estão conectadas por uma estrada em KM como os pesos das arestas. Após fazer a soma do total do primeiro caminho, o próximo caminho (caso houver) que apresentar um total menor, sobreescreve o caminho guardado inicialmente, até que todos os caminhos tenham sido medidos e avaliados.

Os códigos apresentados foram feitos na linguagem Java.

A empresa cobrirá as principais cidades do Nordeste brasileiro.

**-Cidades Atendidas**

* Recife
* Fortaleza
* Salvador
* João Pessoa
* Maceió
* Teresina
* Aracaju
* Natal
* São Luís
* Feira de Santana
* Campina Grande
* Caruaru
* Juazeiro do Norte
* Arapiraca
* Petrolina

1. Grafo escolhido e sua implementação

Para solucionar tal problema, foi implementado um grafo do tipoponderado, onde as cidades são os vértices e os caminhos são as arestas. O grafo direcionado apresenta uma particularidade. Uma aplicação que faz com que cada aresta possua um par ordenado de vértice que se associam.

**Algoritmo de Dijkstra:**

Para sabermos qual o menor caminho, iremos usar o algoritmo de Dijkstra, que analisa qual o menor caminho do ponto de partida até cada vértice do grafo.

Inicialmente, todos os vértices têm uma distância infinita, exceto a cidade de origem, que tem, logicamente, distância zero. Posteriormente, é selecionada a cidade que tem uma distância menor em relação a todas as outras e que se encontra “desmarcada”. A cidade selecionada é “marcada” e são recalculadas, a partir da cidade marcada anteriormente, todas as distâncias às cidades ainda “desmarcadas”, somando à distância da cidade selecionada as distâncias das arestas respectivas. Nos casos em que as novas distâncias obtidas são inferiores às que se encontram armazenadas nas tabelas, procede-se à substituição das mesmas pelas novas distâncias e é alterada também a cidade precedente. De forma análoga, o segundo passo será selecionar a cidade ainda “desmarcada” com a menor distância na tabela, “marcá-la” e, a partir dela, recalcular as distâncias, alterando aquelas que sejam menores que as da tabela. Repete-se o algoritmo até que todas as cidades tenham sido “marcadas”. Quando todas as cidades tiverem sido “marcadas”, os valores obtidos são as distâncias mínimas dos caminhos que partem da cidade inicial para as restantes. O caminho percorrido nesse trajeto é obtido a partir dos valores percorridos e salvos na fila.

A partir disso, o método criado no código fonte tem como base ir atualizando o atributo do vértice chamado distância mínima, que é inicializada infinita, com a soma da distância mínima encontrada no vértice e a distância(peso da aresta) encontrada no vizinho(vértice adjacente ao vértice em execução), até chegar ao destino, porém, o valor da distância mínima só será atualizada quando o valor da soma for inferior ao valor já presente lá no atributo. Na fila de prioridade é armazenado os vértices em operação, e quando o valor é satisfatório para a operação e faz parte do caminho mais curto, o vértice é adicionado nem outra lista, que é para representar o caminho.

Para implementação do grafo foi montado a seguinte classe:

Aresta.java

public class Aresta<TIPO> {

    public *Double* distancia;

    public *Vertice*<TIPO> inicio;

    public *Vertice*<TIPO> fim;

Grafo.java

public class Grafo<TIPO>{

    public final *ArrayList*<*Aresta*<TIPO>> arestasG;

    public final *ArrayList*<*Vertice*<TIPO>> verticesG;

Vertice.java

public class Vertice<TIPO> implements *Comparable*<*Vertice*<TIPO>>{

    public *TIPO* dado;

    public *ArrayList*<*Aresta*<TIPO>> arestaEntrada;

    public *ArrayList*<*Aresta*<TIPO>> arestaSaida;

    public final *ArrayList*<*Aresta*<TIPO>> vizinhos;

    public *LinkedList*<*Vertice*<TIPO>> caminho;

    public *Double* distanciaMin = Double.POSITIVE\_INFINITY;

Dijkstra.java (usada para o método do algoritmo Dijkstra)

public class Dijkstra<TIPO> {

    public *void* menorCaminho(*Vertice*<TIPO> *inicio*) {

        // 1. Guarde o nó que ainda não foi visitado com a distância mínima.

        // 2. Visite todos os vizinhos.

        // 3. Atualize todas das distâncias dos vizinhos(Na fila de prioridade)

        // Repita o processo até que todos os nós sejam visitados.

        inicio.distanciaMin = 0.0;

*PriorityQueue*<*Vertice*<TIPO>> fila = **new** *PriorityQueue*<*Vertice*<TIPO>>();

        fila.add(inicio);

        while (!fila.isEmpty()) {

*Vertice*<TIPO> temp = fila.poll();

            for (*Aresta*<TIPO> vizinho : temp.vizinhos) {

*Double* novaDist = temp.distanciaMin + vizinho.distancia;

                if (vizinho.fim.distanciaMin > novaDist) {

                    // Remove o valor da fila para atualizar a distância.

                    fila.remove(vizinho.fim);

                    vizinho.fim.distanciaMin = novaDist;

                    // Pegue o caminho visitado até agora e adicione o vértice.

                    vizinho.fim.caminho = **new** *LinkedList*<*Vertice*<TIPO>>(temp.caminho);

                    vizinho.fim.caminho.add(temp);

                    //Reinsira o vértice com a nova distância.

                    fila.add(vizinho.fim);

                }

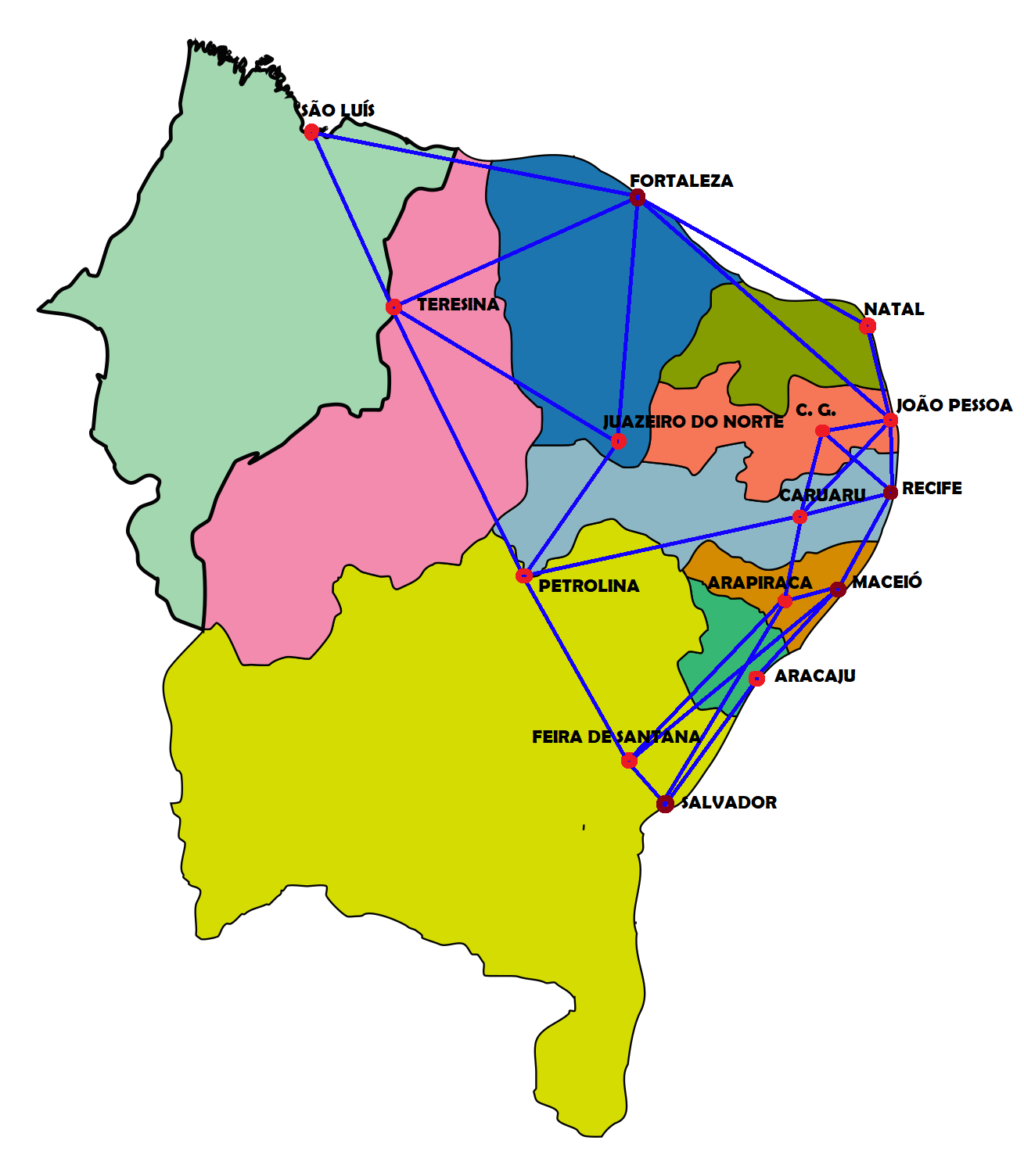
            }

        }

    }

}

1. **Desenho do grafo**



1. Conclusão

Para realizar os teste com o código apresentado, foi montado um grafo a partir da imagem do item 3.

Contendo as seguintes ações na classe Main.java :

System.out.println("\nDIGITE A AÇÃO: \n1 BUSCA EM LARGURA \n2 ROTA MAIS CURTA ENTRE DUAS CIDADES \n3 TODAS AS ROTAS MAIS CURTAS PARTINDO DA CIDADE \n4 DISTÂNCIA DE DETERMINADA ARESTA");

*int* acao = scanInt.nextInt();

        switch (acao) {

            case 1 *->* {

                System.out.println("Digite o Número da cidade que você quer começar: \nRecife: 0\nFortaleza: 1\nSalvador: 2\nJoão Pessoa: 3\nMaceió: 4\nTeresina: 5\nAracaju: 6\nNatal: 7\nSão Luís: 8\nFeira de Santana: 9\nCampina Grande: 10\nCaruaru: 11\nJuazeiro do Norte: 12\nArapiraca: 13\nPetrolina: 14");

*int* index = scanIndex.nextInt();

                grafo.buscaLargura(index);

            }

            case 2 *->* {

                System.out.println("Qual cidade é o ponto de partida?");

*String* cidade = scanCidade.nextLine();

                obj.menorCaminho(grafo.getVertice(grafo.getVertice(cidade).getDado()));

                System.out.println("Destino desejado: ");

*String* dest = scanDestino.nextLine();

                for (*Vertice*<String> rota : grafo.getVerticesG()) {

                    if (rota.getDado().equals(dest)) {

                        System.out.print("Destino - " + rota.getDado() + " , Distância Mínima - " + rota.distanciaMin + "\nCaminho - [ ");

                        for (*Vertice*<String> caminhoV : rota.caminho) {

                            System.out.print(caminhoV.getDado() + " > ");

                        }

                        System.out.println("" + rota.getDado() + " ]");

                    }

                }

            }

            case 3 *->* {

                System.out.println("Você quer ver os caminhos de partindo de que cidade?");

*String* cidade2 = scanCidade.nextLine();

                obj.menorCaminho(grafo.getVertice(grafo.getVertice(cidade2).getDado()));

                for (*Vertice*<String> rota2 : grafo.getVerticesG()) {

                    System.out.print("\nDestino - " + rota2.getDado() + " , Distância Mínima - " + rota2.distanciaMin + "\nCaminho - [ ");

                    for (*Vertice*<String> caminhoV2 : rota2.caminho) {

                        System.out.print(caminhoV2.getDado() + " > ");

                    }

                    System.out.println("" + rota2.getDado() + " ]");

                }

            }

            case 4 *->* {

                System.out.println("Você gostaria de ver a distância(em Km) de que aresta?");

                System.out.println("Início: ");

*String* verticeI = scanCidade.nextLine();

                System.out.println("Destino: ");

*String* verticeF = scanDestino.nextLine();

                if (grafo.getAresta(grafo.getVertice(verticeI), grafo.getVertice(verticeF)) == null) {

                    System.out.println("Essa aresta não existe!");

                } else {

                    System.out.println("A distância entre " + verticeI + " e " + verticeF + " é de: " + grafo.getAresta(grafo.getVertice(verticeI), grafo.getVertice(verticeF)).getDistancia() + " Km");

                }

            }

A fim de teste, obtivemos o seguinte resultado com o algoritmo:

