Instituto Superior de Engenharia do Porto

PROJETO ESINF : PARTE II METRO DE PARIS

Trabalho realizado por:

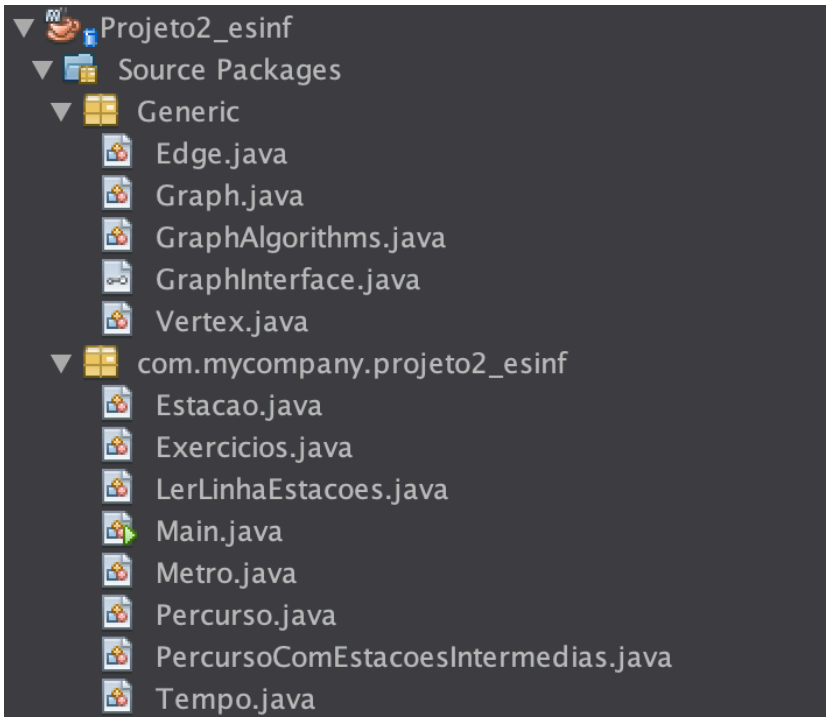
João Flores (1171409);
José Mota (1161263);

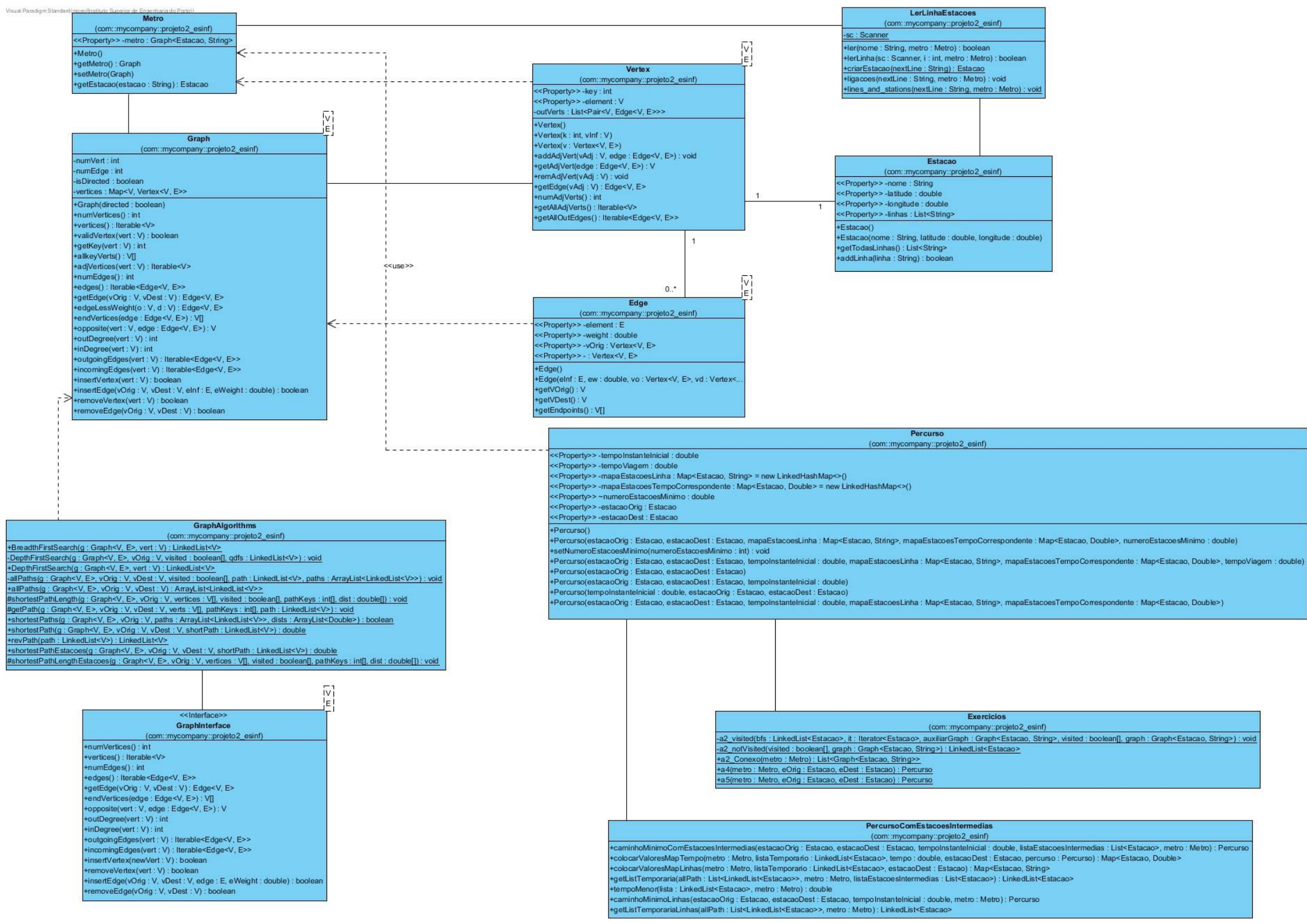
Este projeto insere-se no âmbito na cadeira estruturas de informação com o objetivo de manutenção e gestão da linha de metro de Paris.
De modo a alcançar os objetivos propostos foi realizada uma analise metódica para aplicar uma estrutura correta e ponderar como realizar os objetivos.
Os atributos mais pertinentes é a sugestão de trajetos para os utentes conforme as necessidades destes.

Dada a complexidade da estrutura a utilização de grafos com vértices e ligações é uma estrutura adequada para se puder implementar as necessidades dos utentes.

Para a realização deste projeto, tivemos a necessidade de implementar as seguintes classes :

- Metro
- Estacao
- Connections,LerLinhaEstacoes
- Classes onde foram implementados os requisitos pedidos no enunciado(Exercicios e PercursoComEstacoesIntermedias
- Classes genéricas : Edge,Vertex Graph, GraphAlgorithms
- Tempo





NOTA: O DIAGRMA DE CLASSES SEGUE DENTRO DA PASTA SUBMETIDA UMA VES QUE NÃO SE VÊ NITIDAMENTE!!

Criar o grafo da rede de metro de Paris

A importação de dados é uma necessidade para qualquer projeto deste tipo, logo o ponto inicial do nosso projeto foi a criação de métodos para ler, instanciar e adicionar à estrutura. Foi fornecido três ficheiros com extensão .csv, tipo de ficheiro texto separado por ";" foi criado um objeto do tipo scanner para ler linha a linha até ao fim do ficheiro.
A existência de um método que faz a distinção da instanciação, que vai ser redirecionado conforme o nome do ficheiro, a utilização deste método serve para evitar código repetido.
O método criarEstacao tem como objetivo separar a linha e colocar num vetor de String, a criação da estação é feita ao colocar os elementos do vetor por ordem do construtor. Quando criar a estação retorna-a e no método lerLinha coloca no grafo usando o insertVertex.
Quando as estações já estiverem todas no grafo é necessário colocar todas as linhas que a estação tem. Obtemos a estação pelo segundo parâmetro da linha e adicionamos a uma lista que contem todas as linhas da estação.
Com a utilização de grafos é necessário estabelecer as entre as estações, a ligação do parâmetro um com o parâmetro dois da linha representam respetivamente a estação de origem e destino, o peso entre as duas estações e também a respetiva linha. A utilização do insertEdge para adicionar a ligação entre as duas estações.

Verificar se o grafo é conexo, se não for devolver os seus componentes.

Para este requisito, tivemos de compreender o conceito de conexo. Para ser conexo, um temos de conseguir, através, de um ponto incial, chegar a outro (ponto final), passando por todos os pontos.

No método a2_Conexo(método principal), obtemos a lista de estacoes (LinkedList) através do algoritmo BreadthFirstSearch, as quais consigamos chegar partindo do ponto inicial (obtido através do grafo criado, (vamos buscar todos as keys,ou seja, todas as estações onde a primeira estação se encontra na posição zero do vetor Estacao-- Estacao[] keysEstacao = metro.getMetro().allkeyVerts() Estacao e = keysEstacao[0];).
Seguidamente colocamos num vetor de booleanos (boolean[] visited) se os vértices(Estacoes) já tinham sido ou não visitadas(a2_visited), caso tenha sido visitada inserimos num grafo

```
*/
public class LerLinhaEstacoes {

    private static Scanner sc;
    public boolean ler(String nome, Metro metro) throws FileNotFoundException {
        sc = new Scanner(new File(nome));
        if (nome.equals("connections.csv")) {
            lerLinha(sc, 3, metro);
        }
        if (nome.equals("coordinates.csv")) {
            lerLinha(sc, 1, metro);
        }
        if (nome.equals("lines_and_stations.csv")) {
            lerLinha(sc, 2, metro);
        }

        return true;
    }

    public boolean lerLinha(Scanner sc, int i, Metro metro) {
        if (!sc.hasNext()) {
            return false;
        } else {
            if (i == 1) {
                Estacao estacao = criarEstacao(sc.nextLine());
                metro.getMetro().insertVertex(estacao);
                lerLinha(sc, i, metro);
            }
            if (i == 2) {
                lines_and_stations(sc.nextLine(), metro);
                lerLinha(sc, i, metro);
            }
            if (i == 3) {
                ligacoes(sc.nextLine(), metro);
                lerLinha(sc, i, metro);
            }
            return true;
        }
    }

    public static Estacao criarEstacao(String nextLine) {
        String tmp[] = nextLine.split(";");
        Estacao estacao = null;
        try {
            estacao = new Estacao(tmp[0], Double.parseDouble(tmp[1]), Double.parseDouble(tmp[2]));
        } catch (IllegalArgumentException e) {
            e.getMessage();
        }
        return estacao;
    }

    public void ligacoes(String nextLine, Metro metro) {
        String tmp[] = nextLine.split(";");
        metro.getMetro().insertEdge(metro.getEstacao(tmp[1]), metro.getEstacao(tmp[2]), tmp[0], Double.parseDouble(tmp[3]));
    }

    public static void lines_and_stations(String nextLine, Metro metro) {
        String tmp[] = nextLine.split(";");
        Estacao estacao = metro.getEstacao(tmp[1]);
        if (estacao != null) {
            estacao.addLinha(tmp[0]);
        }
    }
}
```

auxiliar que posteriormente será adicionado a uma list contendo os grafos criados, ou seja, as varias compontes dos grafo.

Posteriormente, vamos buscar as estacoes não visitadas(a2_notVisited) e repetimos o processo, a primeira estacao na linkedList notVisited e utilizamos o BreadthFirstSearch para obter as ligações a essa estacao.e vemos os visitados e acrescescentamos essa estacao ao grafo e no final esse grafo sera a dicionado a list componteees
Caso o tamanho da list compontes seja 1 (apenas contem um grafo) ou se o a linkedList resultante do BreadthFirstSearch contiver todos os vértices, o grafo que contem as estacoes e fortemente conexo, caso contrário lua lista de grafos é retornada.

```
public static List<Graph<Estacao, String>> a2_Conexo(Metro metro) {
    Graph<Estacao, String> graph = metro.getMetro();
    boolean[] visited = new boolean[graph.numVertices()];

    List<Graph<Estacao, String>> componentes = new ArrayList<>();
    Estacao[] keysEstacao = metro.getMetro().allkeyVerts();
    Estacao e = keysEstacao[0];
    LinkedList<Estacao> bfsresult = GraphAlgorithms.BreadthFirstSearch(graph, e);
    if (bfsresult.size() == graph.numVertices() && componentes.isEmpty()) {
        return null;
    } else {
        Graph<Estacao, String> auxiliarGraph = new Graph(true);
        for (Estacao v : bfsresult) {
            visited[graph.getKey(v)] = bfsresult.contains(v);
            auxiliarGraph.insertVertex(v);
        }
        LinkedList<Estacao> notVisited = a2_notVisited(visited, graph);
        Iterator<Estacao> it = notVisited.iterator();
        while (notVisited.size() > 0) {
            a2_visited(bfsresult, it, auxiliarGraph, visited, graph);
            if (!componentes.contains(auxiliarGraph)) {
                componentes.add(auxiliarGraph);
            }

            notVisited = a2_notVisited(visited, graph);
            if (notVisited.size() > 0) {
                auxiliarGraph = new Graph(true);
                Estacao first = notVisited.getFirst();
                auxiliarGraph.insertVertex(first);
                visited[graph.getKey(first)] = true;
                bfsresult = GraphAlgorithms.BreadthFirstSearch(graph, first);
                for (Estacao v : bfsresult) {
                    visited[graph.getKey(v)] = true;
                }
                it = notVisited.iterator();
            }
        }
        if (componentes.size() == 1) {
            return null;
        }
    }
    return componentes;
}
```

```java
private static void a2_visited(LinkedList<Estacao> bfs, Iterator<Estacao> it, Graph<Estacao, String> auxiliarGraph,
        boolean visited[], Graph<Estacao, String> graph) {
    while (it.hasNext()) {
        Estacao estacao = it.next();
        Iterable<Estacao> it1 = graph.adjVertices(estacao);
        for (Estacao adj : it1) {
            if (bfs.contains(adj)) {
                auxiliarGraph.insertVertex(estacao);
                visited[graph.getKey(estacao)] = true;
            }
        }
    }
}

private static LinkedList<Estacao> a2_notVisited(boolean visited[], Graph<Estacao, String> graph) {
    LinkedList<Estacao> notVisited = new LinkedList<>();
    for (Estacao e : graph.vertices()) {
        if (!visited[graph.getKey(e)]) {
            notVisited.add(e);
        }
    }
    return notVisited;
}
```

Dada uma estação origem e uma estação destino encontrar o caminho mínimo entre estas quanto ao número de estações, devolvendo uma instância da classe Percurso.

Para o desenvolvimento deste requisito, necessitamos de uma instância Percurso. Esta classe contém toda a informação relativa a um percurso incluindo as linhas, as estações percorridas em cada linha e o instante em que passa em cada estação, dado um instante inicial do percurso.
Para tal a classe percurso tem um private Map<Estacao, String> mapaEstacoesLinha que contem as estacoes e as respetivas linhas, private Map<Estacao, Double> mapaEstacoesTempoCorrespondente que contem as estacoes e o tempo que demora a chegar a cada estacao que tem ligação, private double tempoInstanteInicial, private double tempoViagem; a estação origem, estacao de origem e o numero mínimo de estacoes.

Tendo a classe Percurso desenvolvida e testada, implementados o requisito solicitado na

```
protected static <V, E> void shortestPathLengthEstacoes(Graph<V, E> g, V vOrig, V[] vertices,
        boolean[] visited, int[] pathKeys, double[] dist) {
    int oKey = g.getKey(vOrig);
    dist[oKey] = 1;

    while (oKey != -1) {
        visited[g.getKey(vOrig)] = true;

        for (Edge<V, E> e : g.outgoingEdges(vOrig)) {
            V vert = e.getVDest() != vOrig ? e.getVDest() : e.getVOrig();
            if (!visited[g.getKey(vert)] && dist[g.getKey(vert)] > dist[g.getKey(vOrig)] + 1.0) {
                dist[g.getKey(vert)] = dist[g.getKey(vOrig)] + 1.0;
                pathKeys[g.getKey(vert)] = g.getKey(vOrig);
            }
        }

        Double min = Double.MAX_VALUE;
        oKey = -1;
        for (int i = 0; i < g.numVertices(); i++) {
            if (!visited[i] && dist[i] < min) {
                min = dist[i];
                oKey = i;
            }
        }
        for (V v : g.vertices()) {
            if (g.getKey(v) == oKey) {
                vOrig = v;
                break;
            }
        }

    }
}
```

```
/**
 *
 * @author j
 */
public class

    private
    private
    private
    private


    // usamo
    private
    private

    private
    private

    double n  }


    private static final double TEMPO_OMISSAO=0;

    public Percurso() {
    }

    public Percurso(Estacao estacaoOrig, Estacao estacaoDest,
            Map<Estacao, String> mapaEstacoesLinha,Map<Estacao,Double> mapaEstacoesTempoCorrespondente,
            double  numeroEstacoesMinimo){

        this.estacaoOrig = estacaoOrig;
        this.estacaoDest = estacaoDest;
        this.numeroEstacoesMinimo=numeroEstacoesMinimo;
        setMapaEstacoesLinha(mapaEstacoesLinha);
        setMapaEstacoesTempoCorrespondente(mapaEstacoesTempoCorrespondente);
    }



    public Percurso(Estacao estacaoOrig, Estacao estacaoDest, double tempoInstanteInicial,
            Map<Estacao, String> mapaEstacoesLinha,
            Map<Estacao, Double> mapaEstacoesTempoCorrespondente,double tempoViagem) {
        this.estacaoOrig = estacaoOrig;
        this.estacaoDest = estacaoDest;
        this.tempoInstanteInicial=tempoInstanteInicial;
        this.tempoViagem=tempoViagem;
        setMapaEstacoesLinha(mapaEstacoesLinha);
        setMapaEstacoesTempoCorrespondente(mapaEstacoesTempoCorrespondente);

    }
```

alínea 4).

Foi recreado o método da classe GraphAlgorithms shortestPath a fim de retornar o número de estações mínima. Assim para esta alínea através do método shortestPathEstacoes obtemos o numero mínimo de estacoes a percorrer desde a estacao de origem até a estacao de destino.

Dada uma estação origem, uma estação destino e um instante inicial encontrar o caminho mínimo entre
estas quanto ao tempo a percorrer, devolvendo uma instância da classe Percurso.

Este requisito é muito semelhante ao anterior. Apenas difere no facto de que o shortestPath retorna o custo mínimo,neste contexto retorna o tempo mínimo desde a estacao inicial ao destino.

```
public static Percurso a4(Metro metro, Estacao eOrig, Estacao eDest) {
    Estacao vOrig=eOrig;
    double peso = 0;
    LinkedList<Estacao> shortstPath = new LinkedList<>();
    double minimoEstacoes = GraphAlgorithms.shortestPathEstacoes(metro.getMetro(), vOrig, eDest, shortstPath);
    Map<Estacao, String> mapEstacaoLinha = new LinkedHashMap<>();
    Map<Estacao, Double> mapaEstacoesTempoCorrespondente = new LinkedHashMap<>();
    for (Estacao e : shortstPath) {
        if (e != vOrig) {
        Edge<Estacao, String> edge = metro.getMetro().getEdge(vOrig, e);
        if(edge!=null){
        String linha = edge.getElement();
        peso += edge.getWeight();
        mapEstacaoLinha.put(e, linha);
        mapaEstacoesTempoCorrespondente.put(e, peso);
        }
    }
     vOrig=e;
       }

    Percurso p = new Percurso(eOrig, eDest, mapEstacaoLinha, mapaEstacoesTempoCorrespondente, minimoEstacoes);
    return p;
}
```

```java
//shortest-path between vOrig and vDest
public static <V, E> double shortestPath(Graph<V, E> g, V vOrig, V vDest, LinkedList<V> shortPath) {
    if (!g.validVertex(vOrig) || !g.validVertex(vDest)) {
        return -1;
    }

    V[] vertices = (V[]) new Object[g.numVertices()];
    int[] pathKeys = new int[g.numVertices()];
    double[] dist = new double[g.numVertices()];
    boolean[] visited = new boolean[g.numVertices()];

    for (int i = 0; i < g.numVertices(); i++) {
        pathKeys[i]=-1;
        visited[i] = false;
        vertices[i] = null;
        dist[i] = Double.MAX_VALUE;
    }

    for (V v : g.vertices()) {
        vertices[g.getKey(v)] = v;
    }

    shortestPathLength(g, vOrig, vertices, visited, pathKeys, dist);
    shortPath.clear();

    if (!visited[g.getKey(vDest)]) {
        return -1;
    }

    getPath(g, vOrig, vDest, vertices, pathKeys, shortPath);

    return dist[g.getKey(vDest)];
}
```

```java
    public static Percurso a5(Metro metro, Estacao eOrig, Estacao eDest) {
        Estacao vOrig=eOrig;
        double instanteInicial=0.0;
        LinkedList<Estacao> shortstPath = new LinkedList<>();
        double minimoTempoEstacoes = GraphAlgorithms.shortestPath(metro.getMetro(), vOrig, eDest, shortstPath);
        double peso = 0;
        Map<Estacao, Double> mapEstacaoPeso = new LinkedHashMap<>();
        Map<Estacao, String> mapEstacaoLinha = new LinkedHashMap<>();
        for (Estacao e : shortstPath) {
            if (e != vOrig) {
                Edge<Estacao, String> edge = metro.getMetro().getEdge(vOrig, e);
                if(edge!=null){
                String linha = edge.getElement();
                peso += edge.getWeight();
                mapEstacaoLinha.put(e, linha);
                mapEstacaoPeso.put(e, peso);
                }
            }
            vOrig = e;
        }
        Percurso p = new Percurso(eOrig, eDest, instanteInicial, mapEstacaoLinha, mapEstacaoPeso, minimoTempoEstacoes);
        return p;
    }

}
```

Para finalizar preenchemos os mapas com a informação relativa as estacoes e a respetiva linha, bem como as estacoes e o tempo que a separa da próxima estacao. A linha conseguimos obter através do edge.getElement() e o peso(tempo que demora a chegar a próxima estacao) através do edge.getWeigth().
Por fim criamos a instância do percurso que contem a estação origem e destino, ,o instante inicial(0.0 segundos), o mapa com as estações e a linha, o mapa com as estações e o tempo até a próxima estacao e o tempo mínimo a percorrer desde a estacai origem a destino .

Dada uma estação origem, uma estação destino, um instante inicial e um conjunto de outras estações intermédias, encontrar o caminho mínimo, quanto ao tempo a percorrer, entre a estação inicial e a final que passe obrigatoriamente por todas as estações intermédias, devolvendo uma instância da classe Percurso.

CaminhoMinimoComEstacoesIntermedias
Neste método é necessário uma estação de origem e destino um tempo inicial, uma lista com as estações que o utente necessita de passar e o grafo. A obtenção de todos os caminhos com as estações de início e fim posteriormente é necessário obter qual dos caminhos anteriores tem o peso menor, para tal é utilizado o método tempoMenor e getListaTemporaria que em conjunto vão retornar a lista que tem menor peso e também todas as estações necessárias, se não existir nenhuma com as estações intermedias o que o método retorna é uma lista vazia. Dado que o retorno é um percurso é criada uma instância do percurso com o tempo inicial, estação origem, estação destino, o tempo de viagem total e os mapas com os tempos e linhas que utiliza os métodos colocarValoresMapTempo e colocarValoresMapLinhas mais abaixo explicados que vão ordenar ambos os mapas conforme os valores da lista.

ColocarValoresMapTempo
A uma lista colocar as estações e o tempo em que se encontrava nessa respetiva estação.
Um ciclo for para obter os edges para poder obter o peso de uma estação até outra e adicionar ao peso que já vinha das estações anteriores.
Finalmente fazer a diferença entre o início e o fim e fazer colocar esse valor no atributo tempo Viagem.

ColocarValoresMapLinha
Mesma logica que o método acima explicado com só que a colocação no map em vez de ser tempo é uma String para indicar a linha.

```java
public Percurso caminhoMinimoComEstacoesIntermedias(Estacao estacaoOrig, Estacao estacaoDest, double tempoInstanteInicial, List<Estacao> listaEstacoesIntermedias, Metro metro) {
    Percurso percurso = new Percurso(tempoInstanteInicial, estacaoOrig, estacaoDest);
    List<LinkedList<Estacao>> allPath = GraphAlgorithms.allPaths(metro.getMetro(), estacaoOrig, estacaoDest);
    if (allPath == null || allPath.isEmpty()) {
        return null;
    }
    LinkedList<Estacao> listaTemporario = getListTemporaria(allPath, metro, listaEstacoesIntermedias);

    Map<Estacao, String> mapaEstacaoLinha = colocarValoresMapLinhas(metro, listaTemporario, estacaoDest);
    percurso.setMapaEstacoesLinha(mapaEstacaoLinha);

    Map<Estacao, Double> mapaEstacoesTempo = colocarValoresMapTempo(metro, listaTemporario, tempoInstanteInicial, estacaoDest, percurso);
    percurso.setMapaEstacoesTempoCorrespondente(mapaEstacoesTempo);

    return percurso;
}

public Map<Estacao, Double> colocarValoresMapTempo(Metro metro, LinkedList<Estacao> listaTemporario, double tempo, Estacao estacaoDest, Percurso percurso) {
    Map<Estacao, Double> mapaEstacoesTempo = new LinkedHashMap<>();
    double tempoInicial = tempo;
    mapaEstacoesTempo.put(listaTemporario.get((listaTemporario.size() - 1)), tempo);
    for (int i = listaTemporario.size() - 1; i > 0; i--) {
        tempo = tempo + (metro.getMetro().getEdge(listaTemporario.get(i - 1), listaTemporario.get(i)).getWeight());
        mapaEstacoesTempo.put(listaTemporario.get(i - 1), tempo);
    }
    percurso.setTempoViagem(tempo - tempoInicial);
    return mapaEstacoesTempo;
}

public Map<Estacao, String> colocarValoresMapLinhas(Metro metro, LinkedList<Estacao> listaTemporario, Estacao estacaoDest) {
    Map<Estacao, String> mapaEstacaoLinha = new LinkedHashMap<>();
    Estacao es = null;
    for (int i = listaTemporario.size() - 1; i > 0; i--) {
        mapaEstacaoLinha.put(listaTemporario.get(i), metro.getMetro().getEdge(listaTemporario.get(i - 1), listaTemporario.get(i)).getElement());
        es = listaTemporario.get(i);
    }
    mapaEstacaoLinha.put(estacaoDest, metro.getMetro().getEdge(es, estacaoDest).getElement());
    return mapaEstacaoLinha;
}

public LinkedList<Estacao> getListTemporaria(List<LinkedList<Estacao>> allPath, Metro metro, List<Estacao> listaEstacoesIntermedias) {
    double peso = Double.MAX_VALUE;
    LinkedList<Estacao> listaTemporario = new LinkedList<>();
    for (LinkedList<Estacao> lista : allPath) {
        double tempoMetodo = tempoMenor(lista, metro);
        System.out.println("Tempo" + tempoMetodo);
        if (lista.containsAll(listaEstacoesIntermedias) && tempoMetodo < peso && tempoMetodo != 0.0) {
            peso = tempoMetodo;
            listaTemporario = lista;
        }
    }
    return listaTemporario;
}

public double tempoMenor(LinkedList<Estacao> lista, Metro metro) {
    double peso = 0.0;
    int tamanho = lista.size();
    for (int i = 0; i <= tamanho - 2; i++) {
        peso = peso + metro.getMetro().getEdge(lista.get(i + 1), lista.get(i)).getWeight();
    }
    return peso;
}
```

Dada uma estação origem e uma estação destino encontrar o caminho mínimo entre estas quanto ao número de mudanças de linha, devolvendo uma instância da classe Percurso.

CaminhoMinimosLinhas

Mesmo raciocínio que o método anterior a variante é o getListTemporariaLinhas que em vez de retornar as com peso menor retorna a lista que passa por menos linhas.

```
    public Percurso caminhoMinimoLinhas(Estacao estacaoOrig, Estacao estacaoDest, double tempoInstanteInicial, Metro metro) {
        Percurso percurso = new Percurso(tempoInstanteInicial, estacaoOrig, estacaoDest);
        List<LinkedList<Estacao>> allPath = GraphAlgorithms.allPaths(metro.getMetro(), estacaoOrig, estacaoDest);
        if (allPath == null || allPath.isEmpty()) {
            return null;
        }
        LinkedList<Estacao> listaTemporario = getListTemporariaLinhas(allPath, metro);
        Map<Estacao, String> mapaEstacaoLinha = colocarValoresMapLinhas(metro, listaTemporario, estacaoDest);
        percurso.setMapaEstacoesLinha(mapaEstacaoLinha);

        Map<Estacao, Double> mapaEstacoesTempo = colocarValoresMapTempo(metro, listaTemporario, tempoInstanteInicial, estacaoDest, percurso);
        percurso.setMapaEstacoesTempoCorrespondente(mapaEstacoesTempo);
        return percurso;

    }

    public LinkedList<Estacao> getListTemporariaLinhas(List<LinkedList<Estacao>> allPath, Metro metro) {
        LinkedList<Estacao> temporaria = new LinkedList<>();
        LinkedList<String> tem = new LinkedList<>();
        int tamanhoTemporaria = Integer.MAX_VALUE;
        for (LinkedList<Estacao> lista : allPath) {
            for (int i = 0; i < lista.size() - 1; i++) {
                String linha = metro.getMetro().getEdge(lista.get(i), lista.get(i + 1)).getElement();
                if (!tem.contains(linha)) {
                    tem.add(linha);
                }
            }
            if (tem.size() < tamanhoTemporaria && !tem.isEmpty()) {
                temporaria = lista;
            }
            tamanhoTemporaria = tem.size();
        }
        return temporaria;
    }

}
```

Analise da Complexidade

Alinea 1)

```
public boolean lerLinha(Scanner sc, int i, Metro metro) {                    3*O(n) + O(1)
    if (!sc.hasNext()) {
      return false;                                                          O(1)
    } else {
      if (i == 1) {
        Estacao estacao = criarEstacao(sc.nextLine());                       O(n)
         metro.getMetro().insertVertex(estacao);                             O(n)
         lerLinha(sc, i, metro);                                             n
      }
      if (i == 2) {
         lines_and_stations(sc.nextLine(), metro);                           2*O(n)
         lerLinha(sc, i, metro);                                             n
      }
      if (i == 3) {
        ligacoes(sc.nextLine(), metro);                                      2*O(n)
         lerLinha(sc, i, metro);                                             n
      }
      return true;                                                           O(1)
```

```
public void ligacoes(String nextLine, Metro metro) {                    2*O(n)

    String tmp[] = nextLine.split(";");                                  O(n)

    metro.getMetro().insertEdge(metro.getEstacao(tmp[1]), metro.getEstacao(tmp[2]), tmp[0],
Double.parseDouble(tmp[3]));                                             O(n)

  }
```

```
    public static Estacao criarEstacao(String nextLine) {                4*O(1)+O(n)

        String tmp[] = nextLine.split(";");                              O(n)

        Estacao estacao = null;                                          O(1)

        try {

          estacao = new Estacao(tmp[0], Double.parseDouble(tmp[1]), Double.parseDouble(tmp[2]));  O(1)

        } catch (IllegalArgumentException e) {

          e.getMessage();                                                O(1)

        }

        return estacao;                                                  O(1)

    }
```

```
public boolean ler(String nome, Metro metro) throws FileNotFoundException {        O(n)+2*O(1)
    sc = new Scanner(new File(nome));                                              O(1)
    if (nome.equals("connections.csv")) {
      lerLinha(sc, 3, metro);                                                      O(n)
    }
    if (nome.equals("coordinates.csv")) {
      lerLinha(sc, 1, metro);
    }
    if (nome.equals("lines_and_stations.csv")) {
      lerLinha(sc, 2, metro);
    }

    return true;                                                                   O(1)
  }
```

Alinea 2)

```
public static List<Graph<Estacao, String>> a2_Conexo(Metro metro) {     O(n^2)

    Graph<Estacao, String> graph = metro.getMetro();                      O(1)
    boolean[] visited = new boolean[graph.numVertices()];                 O(1)

    List<Graph<Estacao, String>> componentes = new ArrayList<>();         O(1)
    Estacao[] keysEstacao = metro.getMetro().allkeyVerts();               O(1)
    Estacao e = keysEstacao[0];                                           O(1)
                        LinkedList<Estacao> bfsresult
                        GraphAlgorithms.BreadthFirstSearch(graph, e);     O(1001)
 if (bfsresult.size() == graph.numVertices() && componentes.isEmpty()) {  O(1)
        return null;
                                                                          O(1)

     } else {                                                             O(1)
        Graph<Estacao, String> auxiliarGraph = new Graph(true);           O(1)
        for (Estacao v : bfsresult) {                                     O(n)
           visited[graph.getKey(v)] = bfsresult.contains(v);              O(1)
           auxiliarGraph.insertVertex(v);                                 O(1)
        }
        LinkedList<Estacao> notVisited = a2_notVisited(visited, graph);   O(n^2)
        Iterator<Estacao> it = notVisited.iterator();                     O(1)
        while (notVisited.size() > 0) {                                   O(1)
           a2_visited(bfsresult, it, auxiliarGraph, visited, graph);      O(n^2)
           if (!componentes.contains(auxiliarGraph)) {                    O(1)
              componentes.add(auxiliarGraph);                             O(1)
           }

           notVisited = a2_notVisited(visited, graph);                    O(n^2)
           if (notVisited.size() > 0) {                                   O(1)
              auxiliarGraph = new Graph(true);                            O(1)
              Estacao first = notVisited.getFirst();                      O(1)
              auxiliarGraph.insertVertex(first);                          O(1)
              visited[graph.getKey(first)] = true;                        O(1)
              bfsresult = GraphAlgorithms.BreadthFirstSearch(graph, first);  O(1011)
              for (Estacao v : bfsresult) {                               O(n)
                 visited[graph.getKey(v)] = true;                         O(1)
              }
              it = notVisited.iterator();                                 O(1)
           }
        }
        if (componentes.size() == 1) {                                    O(1)
           return null;                                                   O(1)
        }
     }
     return componentes;                                                  O(1)
  }
```

```
private static void a2_visited(LinkedList<Estacao> bfs, Iterator<Estacao> it,
Graph<Estacao, String> auxiliarGraph, boolean visited[], Graph<Estacao, String>
graph) {                                                                    O(n²)

 while (it.hasNext())
{                                                                  O(n)

        Estacao estacao =
it.next();                                                  O(1)
        Iterable<Estacao> it1 =
graph.adjVertices(estacao);                                 O(1)
        for (Estacao adj : it1)
{                                                           O(n)
            if (bfs.contains(adj)) {                                        O(1)
               auxiliarGraph.insertVertex(estacao);                         O(1)
               visited[graph.getKey(estacao)] =
true;                                       O(1)
            }
        }
    }
  }
```

```
  private static LinkedList<Estacao> a2_notVisited(boolean visited[], Graph<Estacao,
String> graph) {                                                            O(n²)

    LinkedList<Estacao> notVisited = new LinkedList<>
();                                  O(1)
    for (Estacao e : graph.vertices()) {                                    O(n)
       if (!visited[graph.getKey(e)])
{                                                   O(1)

 notVisited.add(e);                                                 O(1)
       }
    }
    return notVisited;                                                  O(1)
  }
```

Alinea 4)

```
  public static Percurso a4(Metro metro, Estacao eOrig, Estacao eDest) {          O(n)

      Estacao vOrig=eOrig;                                                          O(1)
      double peso = 0;                                                              O(1)
      LinkedList<Estacao> shortstPath = new LinkedList<>();                        O(1)
      double minimoEstacoes =
                                                                                    GraphAlgorith
ms.shortestPathEstacoes(metro.getMetro(), vOrig, eDest, shortstPath);
                                                                                    O(1011)
      Map<Estacao, String> mapEstacaoLinha = new LinkedHashMap<>
();                  O(1)
      Map<Estacao, Double> mapaEstacoesTempoCorrespondente = new
LinkedHashMap<>();
                                                                                    O(1)
      for (Estacao e : shortstPath)
{                                                                                   O(n)
          if (e != vOrig)
{                                                                                   O(1)
          Edge<Estacao, String> edge = metro.getMetro().getEdge(vOrig, e);          O(1)
          if(edge!=null)
{                                                                                   O(1)
          String linha =
edge.getElement();                                                                  O(1)
          peso +=
edge.getWeight();                                                                   O(1)
          mapEstacaoLinha.put(e, linha);                                            O(1)
          mapaEstacoesTempoCorrespondente.put(e,
peso);                                    O(1)
          }
      }

vOrig=e;                                                                            O(1)
        }

      Percurso p = new Percurso(eOrig, eDest, mapEstacaoLinha,
mapaEstacoesTempoCorrespondente,
minimoEstacoes);                                  O(1)
      return
p;                                                                                  O(1)
  }
```

Alinea 5)

```
  public static Percurso a5(Metro metro, Estacao eOrig, Estacao eDest) {     O(1)

     Estacao vOrig=eOrig;                                                    O(1)
     double instanteInicial=0.0;                                            O(1)
     LinkedList<Estacao> shortstPath = new LinkedList<>();                  O(1)
     double minimoTempoEstacoes = GraphAlgorithms.shortestPath(metro.getMetro(),
vOrig, eDest, shortstPath); O(1011)
     double peso = 0;                                                        O(1)
     Map<Estacao, Double> mapEstacaoPeso = new LinkedHashMap<>();    O(1)
     Map<Estacao, String> mapEstacaoLinha = new LinkedHashMap<>();
O(1)
     for (Estacao e : shortstPath) {                                        O(n)
        if (e != vOrig) {                                                    O(1)
           Edge<Estacao, String> edge = metro.getMetro().getEdge(vOrig,
e);          O(1)
           if(edge!=null){                                                   O(1)
           String linha =
edge.getElement();                                          O(1)
           peso +=
edge.getWeight();                                                O(1)
           mapEstacaoLinha.put(e,
linha);                                              O(1)
           mapEstacaoPeso.put(e, peso);                                      O(1)
           }
        }
        vOrig = e;                                                           O(1)
     }
     Percurso p = new Percurso(eOrig, eDest, instanteInicial, mapEstacaoLinha,
mapEstacaoPeso,
minimoTempoEstacoes);                                           O(1)
     return
p;                                                                          O(1)
  }

}
```

Alinea 6) e 7)

```
public Percurso caminhoMinimoComEstacoesIntermedias(Estacao estacaoOrig, Estacao estacaoDest, double
tempoInstanteInicial, List<Estacao> listaEstacoesIntermedias, Metro metro) {          3*O(n)^2=O(n)^2

Percurso percurso = new Percurso(tempoInstanteInicial, estacaoOrig, estacaoDest);          O(1)
List<LinkedList<Estacao>> allPath = GraphAlgorithms.allPaths(metro.getMetro(), estacaoOrig, estacaoDest);

                                                                                          O(n)

    if (allPath == null || allPath.isEmpty()) {

      return null;                                                                        O(1)

     }

LinkedList<Estacao> listaTemporario = getListTemporaria(allPath, metro, listaEstacoesIntermedias);  O(n)^2

Map<Estacao, String> mapaEstacaoLinha = colocarValoresMapLinhas(metro, listaTemporario, estacaoDest);
                                                                                          O(n)^2

percurso.setMapaEstacoesLinha(mapaEstacaoLinha);                                          O(1)

Map<Estacao, Double> mapaEstacoesTempo = colocarValoresMapTempo(metro, listaTemporario,
tempoInstanteInicial, estacaoDest, percurso);                                             O(n)^2

percurso.setMapaEstacoesTempoCorrespondente(mapaEstacoesTempo);                           O(1)

return percurso;                                                                          O(1)

  }
```

```
public Map<Estacao, String> colocarValoresMapLinhas(Metro metro, LinkedList<Estacao> listaTemporario,
Estacao estacaoDest) {                                                    O(n)^2+4*O(1)=O(n)^2

    Map<Estacao, String> mapaEstacaoLinha = new LinkedHashMap<>();                O(1)

     Estacao es = null;                                                           O(1)

    for (int i = listaTemporario.size() - 1; i > 0; i--) {                        O(n)
mapaEstacaoLinha.put(listaTemporario.get(i), metro.getMetro().getEdge(listaTemporario.get(i - 1),
listaTemporario.get(i)).getElement());                                            O(n)

       es = listaTemporario.get(i);                                               O(1)

    }

    mapaEstacaoLinha.put(estacaoDest, metro.getMetro().getEdge(es, estacaoDest).getElement()); O(1)

     return mapaEstacaoLinha;                                                     O(1)

}
```

```
public Map<Estacao, Double> colocarValoresMapTempo(Metro metro, LinkedList<Estacao> listaTemporario,
double tempo, Estacao estacaoDest, Percurso percurso) {
```

5*O(1)+O(n)^2=O(n)^2

```
    Map<Estacao, Double> mapaEstacoesTempo = new LinkedHashMap<>();                    O(1)

    double tempoInicial = tempo;                                                       O(1)

    mapaEstacoesTempo.put(listaTemporario.get((listaTemporario.size() - 1)), tempo);   O(1)

    for (int i = listaTemporario.size() - 1; i > 0; i--) {                             O(n)

      tempo = tempo + (metro.getMetro().getEdge(listaTemporario.get(i - 1),
          listaTemporario.get(i)).getWeight());                                        O(n)

      mapaEstacoesTempo.put(listaTemporario.get(i - 1), tempo);                        O(1)

}

    percurso.setTempoViagem(tempo - tempoInicial);                                     O(1)

    return mapaEstacoesTempo;                                                          O(1)

 }
```

```
public Percurso caminhoMinimoLinhas(Estacao estacaoOrig, Estacao estacaoDest, double tempoInstanteInicial,
Metro metro) {                                                                    3*O(n)^2+4*O(1)=O(n)^2

    Percurso percurso = new Percurso(tempoInstanteInicial, estacaoOrig, estacaoDest);                O(1)

    List<LinkedList<Estacao>> allPath = GraphAlgorithms.allPaths(metro.getMetro(), estacaoOrig, estacaoDest);

     if (allPath == null || allPath.isEmpty()) {

       return null;                                                                                  O(1)

     }

    LinkedList<Estacao> listaTemporario = getListTemporariaLinhas(allPath, metro);          O(n)^2

    Map<Estacao, String> mapaEstacaoLinha = colocarValoresMapLinhas(metro, listaTemporario, estacaoDest);
                                                                                                    O(n)^2

    percurso.setMapaEstacoesLinha(mapaEstacaoLinha);                                            O(1)


     Map<Estacao, Double> mapaEstacoesTempo = colocarValoresMapTempo(metro, listaTemporario,
tempoInstanteInicial, estacaoDest, percurso);                                                   O(n)^2

    percurso.setMapaEstacoesTempoCorrespondente(mapaEstacoesTempo);                             O(1)

     return percurso;                                                                           O(1)


  }
```

```
public double tempoMenor(LinkedList<Estacao> lista, Metro metro) {          O(n)+3*O(1)=O(n)
    double peso = 0.0;                                                      O(1)
    int tamanho = lista.size();                                             O(1)
    for (int i = 0; i <= tamanho - 2; i++) {                                O(n)
      peso = peso + metro.getMetro().getEdge(lista.get(i + 1), lista.get(i)).getWeight();   O(1)
    }
    return peso;                                                            O(1)
  }
```

```
public LinkedList<Estacao> getListTemporariaLinhas(List<LinkedList<Estacao>> allPath, Metro metro) {
                                                                                    O(n)^2+4*O(1)
    LinkedList<Estacao> temporaria = new LinkedList<>();                            O(1)
    LinkedList<String> tem = new LinkedList<>();                                    O(1)
     int tamanhoTemporaria = Integer.MAX_VALUE;                                     O(1)
    for (LinkedList<Estacao> lista : allPath) {                                     O(n)
       for (int i = 0; i < lista.size() - 1; i++) {                                 O(n)
         String linha = metro.getMetro().getEdge(lista.get(i), lista.get(i + 1)).getElement();          O(1)
        | if (!tem.contains(linha)) {
            tem.add(linha);                                                         O(1)
        }    }
       if (tem.size() < tamanhoTemporaria && !tem.isEmpty()) {
         temporaria = lista;                                                        O(1)
       }
       tamanhoTemporaria = tem.size();                                              O(1)
    }
    return temporaria;                                                              O(1)
  }
```

```
public LinkedList<Estacao> getListTemporaria(List<LinkedList<Estacao>> allPath, Metro metro, List<Estacao>
listaEstacoesIntermedias) {                                                    3*O(1)+O(n)^2=O(n)^2

    double peso = Double.MAX_VALUE;                                                    O(1)

    LinkedList<Estacao> listaTemporario = new LinkedList<>();                          O(1)

    for (LinkedList<Estacao> lista : allPath) {                                        O(n)

        double tempoMetodo = tempoMenor(lista, metro);                                 O(n)

        if (lista.containsAll(listaEstacoesIntermedias) && tempoMetodo < peso && tempoMetodo != 0.0) {

            peso = tempoMetodo;                                                        O(1)

            listaTemporario = lista;                                                   O(1)

        }

    }

    return listaTemporario;                                                            O(1)

}
```

Possiveis Melhoramentos:

Um aspeto a melhorar em futuros trabalhos é a implementação de estruturas mais otimizadas e eficientes.