Relatório do Projeto A2 - PAA

Guilherme Moreira Castilho,
Paulo César Gomes Rodrigues,
Pedro Santos Tokar,
Vitor Matheus do Nascimento Moreira

5 de dezembro de 2024

1 Introdução

1.1 Apresentação do problema

O presente trabalho tem como proposta auxiliar a cidade fictícia de Vargas à planejar suas linhas de transporte público (metrôs e ônibus) e desenvolver um serviço para encontrar rotas eficientes utilizando-se dos meios de transporte disponíveis para ir de um local a outro dentro da cidade.

As 3 tarefas a serem desenvolvidas para este projeto são:

- Projeto das linhas de metrô da cidade;
- Projeto da linha de ônibus *hop-on/hop-off* da cidade;
- Criação de um serviço para fornecer a rota mais rápida entre dois endereços utilizando os serviços da cidade.

2 Modelagem Arquitetural da Solução e Estruturas de Dados Criadas

2.1 Geração da Planta da Cidade

A planta da cidade de Vargas foi modelada como um único grafo. Os **vértices** desse grafo representam os **cruzamentos** da malha de ruas da cidade, e as **arestas** representam os **segmentos de rua** que ligam esses cruzamentos. As **ruas**, devido à essa natureza, são conjuntos de arestas contínuas que formam caminhos simples no grafo (cada aresta sinaliza a que rua ela pertence). As **regiões** da cidade também são conjuntos de arestas contínuas do grafo, mas dessa vez não formando caminhos simples. Detalhamentos de que informações cada componente do grafo carrega são fornecidas abaixo.

Para gerar grafos de exemplo que seguem essas especificações, utilizou-se da linguagem Python. Todo o código utilizado pode ser visto em `extras/graphgen/graphgen.py`. Por fugir do escopo principal do trabalho, esse módulo não será detalhado no relatório. A implementação consiste em uma classe geradora de grafos com vários métodos para gerar a cidade e salvar seus dados em um arquivo do tipo `.csv`.

Após este ser gerado, desenvolveu-se um script **C++** (presente em `src/cityParser.cpp`) que tem como objetivo inserir todas as informações do `.csv` às estruturas de dados desenvolvidas em **C++** e assim iniciar o desenvolvimento dos algoritmos para resolução de problemas.

2.2 Arestas / Segmentos de Rua

Para modelar as arestas do grafo, criou-se uma estrutura de dados StreetSegment que possui os seguintes atributos:

- lenght: Representa o tamanho do segmento;
- maxSpeed: Velocidade máxima permitida no segmento;
- escavationCost; Custo para escavação de um metrô sob aquele segmento;
- nResidential, nComercial, nIndustrial e nTouristic: Número de imóveis em dado segmento que são de tipo residencial, comercial, industrial e turístico respectivamente;
- street: Inteiro que representa de qual rua o segmento faz parte;
- streetOffset: Intervalo da rua que o segmento faz parte (Para cálculo do número dos imóveis)
- region: Região da cidade em que o segmento está localizado.

2.3 Vértices / Cruzamentos

Para os vértices, foi criado um struct Crossing que contém somente a região tal que o vértice pertence e os segmentos que são incidentes a ele.

2.4 Grafo / Planta da Cidade

A implementação do grafo foi realizada com base no modelo de listas de adjacências, tendo em vista os algoritmos pensados para cada problema e sua complexidade nesse modelo. Apenas em um momento se fez necessário ter um grafo representado por uma matriz de adjacências, e nesse caso a matriz foi feita sem uma classe.

Para a definição da planta da cidade, foi criada uma classe com uma série de parâmetros que eram pertinentes para o problema. Dentre eles estão:

- m_numVertices, m_numRegions e m_numEdges: Contadores para obter as informações respectivamente de número de vértices, regiões e arestas para eventuais cálculos;
- m_vertices e regions: lista de vértices e regiões no grafo;

Além disso, a classe também conta com diversos métodos para a realização de várias atividades:

- hasSegment: Checa se dois vértices possuem conexão em $O(V)$;
- addSegment: Conecta dois vértices em $O(V)$;
- removeSegment: Desconecta dois vértices em $O(V)$;
- print: Permite que o grafo seja visualizado em $O(V + E)$;
- isSubGraph: Verifica se um grafo está contido em outro em $O(E + E')$, sendo E' o número de arestas do subgrafo;
- isValidPath: Determina se um caminho é válido em $O(nV)$ sendo n o número de vértices no caminho;
- hasPath: Verifica se existe um caminho entre dois cruzamentos em $O(V + E)$
- CPTDijkstra e CPTDijkstraRegion: utilizam o algoritmo de Dijkstra para computar o menor caminho entre dois vértices.

2.5 Outras Estruturas de Dados

- Hashtable: implementada utilizando-se listas encadeadas. Operações implementadas incluem inserção, busca, remoção e exibição. Por utilizar listas encadeadas, esta poderia alcançar, no pior caso, complexidade $O(V)$. No presente trabalho, a hashtable foi utilizada no armazenamento de vértices presentes em cada rua, e como as ruas foram definidas de forma a ser impossível possuir mais que um número c de vértices, é possível definir um tamanho de conjunto $M > c$, de forma que colisões são evitadas.

- Heap: Implementado para otimizar atividades que dependem de uma fila de prioridade, tais como Dijksta e Prim. Heap possui métodos implementados como: Inserção ($O(\log n)$), remoção ($O(\log n)$) e extração do topo ($O(\log n)$), Heapify ($O(\log n)$) e impressão ($O(n)$). Por ser usada nos vértices dos grafos, ela realiza as comparações baseando-se em um vetor de custos, e é capaz de corretamente atualizar elementos caso estes já estejam inseridos (em nenhum contexto era necessário ter um elemento inserido duas vezes na heap).

- Lista Encadeada: Listas encadeadas são uma das estruturas de dados mais utiizadas durante o projeto, pois esta serve tanto para a produção de algoritmos, quanto para o desenvolvimento de outras estruturas de dados. Além de iteradores, a estrutura possui métodos para adição e impressão de elementos em $O(n)$

3 Algoritmos Base

Os algorítmos apresentados nessa sessão são usados em mais de um problema, ou seja, são peças fundamentais para construir os algorítmos que resolvem os problemas.

3.1 Dijkstra

O algorítmo de Dijkstra foi implementado usando a implementação eficiente com heaps vista em sala de aula. Em diferentes momentos do trabalho é necessário fazer comparações com diferentes atributos das arestas, e por isso a implementação feita é capaz de receber uma função que diz qual atributo da aresta deve ser usado como base de comparação. Isso permitiu mais flexibilidade e menos código repetido no trabalho.

```
def Dijkstra(v0, Grafo)
    # Inicializacões
    visitados[Número de Vértices do Grafo]
    pais[Número de Vértices do Grafo]
    distancias[Número de Vértices do Grafo]

    para cada vértice i
        pais[i] ← -1
```

```
        distancias[i] ← INFINITO
        visitados[i] ← false

    # Caso Base
    distancias[v0] ← 0
    pais[v0] ← v0

    # Inicializacão do Heap
    heap ← Heap(Tamanho: V vértices, Tipo: Mínimo)
    heap.insert(v0)

    enquanto heap não vazio:
        v1 ← heap.popTop()
        se distancias[v1] igual INFINITO então paramos loop

        para cada aresta na lista de adjacência de v1:
            v2 ← outra ponta da aresta
            se v2 não foi visitado:
                se distancias[v1] + peso da aresta < distancia[v2]:
                    distancias[v2] ← distancias[v1] + peso da aresta
                    pais[v2] ← v1
                    heap.insert(v2)

        visitados[v1] ← true
```

Como cada vértice é avaliado uma única vez (V); A operação de remover o elemento v_i do topo da heap tem complexidade $O(logV)$; O relaxamento é feito em cada aresta (E), e vértices podem ser inseridos na heap com complexidade $O(logV)$.

A complexidade final é $O((V+E)logV)$, e por se tratar de grafos exparsos temos uma complexidade $O(VlogV)$.

Uma variante desse algorítmo que não verifica arestas que não pertencem à uma região específica também foi feita, para auxiliar na criação dos sistemas de metrô e ônibus.

3.2 Prim

```
def genMSTPrim(grafo)
    mst ← Grafo vazio com os mesmos vértices de grafo

    # Inicializacões2
    distâncias[Número de Vértices]
    visitados[Números de vértices]
    pais[Número de vértices]

    para cada vértice i:
        distâncias[i] ← INFINITO
        visitados[i] ← false
        pais[i] ← -1

    # Caso base
    distâncias[0] ← 0;
    pais[0] ← 0

    # Inicializacão do Heap
    heap ← Heap(Tamanho: V vértices, Tipo: Mínimo)
    heap.insert(v0)

    enquanto heap não vazio:
        v1 ← heap.popTop()
        visitados[v1] ← true

        para cada aresta na lista de adjcência de v1:
            v2 ← Outra ponta da aresta
            se v2 não visitado e tamanho da aresta < distâncias[v2]:
                distancias[v2] ← tamanho da aresta
                pais[v2] ← v1
                heap.insert(v2)

    # Construindo a árvore
    para cada vértice v do grafo:
        se pais[v] diferente de -1:
            adicionamos ambas as arestas à mst conectando v1 a seu pai e vice-versa.
```

```
            aresta1, aresta2 ← arestas com tamanho
   distâncias[v]

    retornar mst
```

Como cada vértice é avaliado uma única vez (V): em cada avaliação procura o vértice na fronteira com menor custo de adição à árvore ($O(logV)$) e atualiza a fronteira nas $|E|$ arestas.

A complexidade é $O((V+E)logV)$, e como $E \leq V^2$ (o grafo da cidade é esparso), a complexidade do algorítmo é $O(ElogV)$.

4 Tarefa 1: Projeto das linhas de metrô da cidade

Deve-se projetar um algoritmo capaz de definir as linhas de metrô de forma que o custo para a cidade seja mínimo e as estações estejam devidamente conectadas.

As estações devem estar localizadas no cruzamento que minimiza a maior distãncia deste cruzamento ao ponto mais distante de sua região e todas as regiões necessitam obrigatoriamente possuir uma estação de metrô.

O algoritmo desenvolvido para resolver esse problema consiste em:

```
def genSubwayStation(Grafo, região, estacões[])
    # Inicializacões
    maxDistâncias[Número de Vértices]
    distâncias[Número de Vértices] # Será atualizado
   por Dijkstra abaixo

    para cada vértice i:
        maxDistàncias[i] ← -1

    para cada vértice v1 na região:
        Dijkstra(v1, distâncias, região) # Dijkstra
   levando em consideracão uma região e distâncias como
   peso
        para cada vértice w na região:
            se distâncias[v2] > maxDistâncias[v2]:
```

```
                maxDistâncias[v2] ← distâncias[v2]

    minValor ← INFINITO
    melhorVértice ← -1

    para cada vértice v na região:
        se maxDistâncias[v] < minValor:
            minValor ← maxDistâncias[v]
            melhorVértice ← v

    estacões[region] = melhorVértice
```

Essa rotina determina, para uma região específica, qual vértice deve receber uma estação de metrô. Para fazer isso, é necessário encontrar a CPT para cada vértice, e armazenar qual a maior distância existente na CPT. Como o algorítmo de Dijkstra é executado para cada vértice, essa etapa tem complexidade $O(V_R^2 log V_R)$, sendo V_R a quantidade de vértices da região.

O resultado dessa etapa é uma array que relaciona cada vértice com sua distância máxima, bastando apenas selecionar o menor elemento dessa array para completar a escolha. A etapa de encontrar as maiores distâncias domina essa rotina, e sua complexidade é $O(V_R^2 log V_R)$.

Como isso é executado para todas as regiões, a complexidade é $O(\sum_i^R V_{R_i}^2 log V_{R_i})$. Como $f(x) = x^2 log x$ é uma função convexa, temos pela desigualdade de Jensen que

$$f(\frac{1}{n}\sum_{i=1}^{n} x_i) \leq \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

Aplicando para Jensen o nosso problema:

$$\sum_i^n V_{R_i}^2 log V_{R_i} \leq R(\frac{1}{R}\sum_{i=1}^{R} V_{R_i})^2 log(\frac{1}{R}\sum_{i=1}^{R} V_{R_i})$$

$$\sum_i^n V_{R_i}^2 log V_{R_i} \leq R(\frac{1}{R}V)^2 log(\frac{1}{R}V)$$

$$\sum_i^n V_{R_i}^2 log V_{R_i} \leq \frac{V^2}{R} log(\frac{V}{R})$$

Portanto a complexidade de *genSubwayStation* é $O(\frac{V^2}{R} log\frac{V}{R})$.

As linhas de metrô devem ser escavadas sob as ruas da cidade, tendo cada segmento da rua um custo para escavá-lo e deve ser possível ir qualquer estação à outra. Tendo esses critérios em vista, o algorítmo que escolhe quais estações estarão conectadas consiste em:

1 - Calcular, para cada região, a distância de sua estação para as outras estações, com Dijkstra

2 - Montar um grafo completo com essas distâncias

3 - Encontrar a MST desse grafo completo.

O pseudocódigo é:

```
def getSubwayLines(grafo, estacões[], lista de R caminhos vazios)
    define grafoCompleto #os vértices representam as regiões da cidade
    custo[Número de vértices do grafo]
    para cada região r da cidade:
        Dijkstra(grafo, estacões[r], caminhos[r]) # Dijkstra levando o Custo de Escavacão como peso
        para cada região s da cidade:
            - se r igual a s, continua para o pr ximo s
            aresta = nova aresta com custo cost[stations[s]]
            Adicionamos a aresta ao grafoCompleto a conectando a *r* e *s*

    metrôMST = genMSTPrim(grafoCompleto)
    retornar metrôMST
```

O cáculo da distância de uma estação para as outras é feito executando um Dijkstra no grafo da cidade ($O(VlogV)$ para cada iteração) e atualiza o custo para as demais regiões. Por fim, fora dos loops, executa uma MST Prim para o grafo completo, que tem vértices correspondentes às regiões. Essa última etapa tem complexidade ($O(R^2 logR)$), já que esse grafo é denso.

Por fim temos que sua complexidade será $O(R(VlogV + R) + VlogV) = O(RVlogV)$.

Tendo a MST e as CPTs de cada estação armazenadas, é possível construir um subgrafo do grafo da cidade com complexidade ($O(RV^2)$) (temos R

- 1 ligações entre os metrôs, e fazer cada ligação tem complexidade $O(V^2)$).

Como a cidade só terá que executar eses algorítmo apenas uma vez, as complexidades são aceitáveis para a solução e o contexto.

5 Tarefa 2: Projeto das linhas de ônibus da cidade

A linha de ônibus deve ser projetada de forma a haver somente uma linha que passe por todas as regiões da cidade iniciando e terminando no mesmo lugar. Por ser somente uma linha, é desejado maximizar o número de imóveis comerciais e atrações turísticas.

O problema apresentado aqui é bem semelhante ao Problema do Caixeiro Viajante: é necessário encontrar um ciclo que tenha "custo"mínimo. Para adequar possíveis soluções do Problema do Caixeiro Viajante no contexto de Vargas, as seguintes decisões foram tomadas:

Como é necessário que o ônibus passe uma vez em cada região, foi decidido que em cada região seria selecionado um cruzamento que o ônibus **deve** passar. Essa seleção garante que o ônibus não irá ignorar uma região, e torna possível usar esses pontos obrigatórios como um grafo para aplicar soluções do PCV. Como a seleção de pontos e as soluções do PCV dependem de pesos para as arestas, adotamos o seguinte cálculo para realizar as operações de comparação:

$$peso = \left(\frac{N_{industrial} + N_{residencial}}{O_{comercial} + O_{turístico}} \right) N_{construções}$$

Como se trata de uma razão entre construções indesejadas na rota do ônibus e construções desejadas, valores mais altos dos pesos significarão ruas "piores"e evitadas pelos algorítmos já conhecidos. É feita uma multiplicação ao final pelo número total de construções para que o tamanho das ruas seja levado em conta.

Vale aqui fazer uma diferenciação: esses pontos por região **não** são a mesma coisa que pontos de ônibus. Na solução da tarefa 3, qualquer cruzamento que o ônibus passa é um ponto de ônibus que permite subida e descida. Esses pontos são usados apenas na construção da linha.

A seleção dos pontos dentro das regiões segue a mesma lógica da seleção das estações do metrô. Como minimizar a distância máxima é de certa forma uma maneira de encontrar o ”centro”da região, esse foi o critério mais adequado para selecionar os pontos.

```
def genBusPoints(city, região, points[])
    maxDist[Número de Vértices da cidade]
    distâncias[Número de vértices da cidade]
    para cada i em número de vértices:
        maxDist[i] = -1

    para cada vértice v1 da região:
        Dijkstra(v1, distâncias, região) # Dijkstra levando em consideracão uma região e razão [(industrial+residencial)/(comercial+turístico + 1] como peso
        para cada vértice v2 da região:
            se distâncias[v2] > maxDist[v2]:
                maxDist[v2] ← distâncias[v2]

    minValue ← INFINITO
    melhorVértice ← -1

    para cada vértice v da região:
        se maxDist[v] < minValue:
            minValue ← maxDist[v]
            melhorVértice ← v

    points[region] ← melhorVértice
```

Essa rotina é semelhante à *genSubwayStations*, e apresenta a mesma complexidade: $O(\frac{V^2}{R} log \frac{V}{R})$. A principal diferença entre as duas está na operação feita para determinar o peso de uma aresta.

A próxima etapa também é bem semelhante ao que é feito na seleção das linhas de metrô: um grafo completo é construído, para seleção de quais regiões terão ligações. Dessa vez, o grafo é representado em forma de matriz de adjacências, já que consultar a distância entre os pontos de duas regiões será uma operação muito utilizada.

```
def genBusLinesFull(city, points[], distMatrix, path)
    cost[número de vértices da cidade]
```

```
    para cada região i da cidade:
        Dijkstra(grafo, points[r], path[r], cost) #
    Dijkstra levando a razão
    [(industrial+residencial)/(comercial+turístico + 1]
    como peso das arestas
        para cada região j da cidade:
            se i igual a j:
                pular a iteracão
            distMatrix[i][j] = cost[points[j]]
```

Complexidade: Para cada região executa um Dijkstra por todo grafo e calcula a distância do ponto da região atual com os das demais regiões. Com isso temos $O(R(VlogV + R)) = O(RVlogV)$.

O PCV não tem solução exata. Para conseguir uma boa aproximação, são feitas duas etapas:

1 - Construção de um ciclo inicial usando um algorítmo guloso. Dado um vértice de ínicio v, o caminho é construído sempre indo do último vértice para o vértice mais próximo não visitado.

```
def genBusLines(numRegions, busLine[], distMatrix[][])
    visitados[Número de Regiões]
    para cada i em número de regiões:
        visitados[i] ← false

    v ← 0
    busline[v] ← 0
    visitados[v] ← true

    totalDist ← 0
    ContadorArestas ← 0

    enquanto ContadorArestas diferente de numRegions-1:
        melhorV ← -1
        minDist ← INFINITO

        para cada região r do grafo:
            se visitados[r] falso e distMatrix[v][r] <
    minDist:
```

```
                minDist ← distMatrix[v][r]
                melhorV ← r

        busline[ContadorArestas + 1] ← melhorV
        visitados[melhorV] ← true
        ContadorArestas += 1
        v = melhorV
        totalDist += minDist

    retornar totalDist
```

Complexidade: No pior caso, o loop while percorrerá um valor proporcional à R vértices, e para cada iteração do loop, é passado por cada região. Com isso temos uma complexidade $O(R^2)$.

2 - Aprimoramento do ciclo. O aprimoramento consiste em analisar todos os pares de arestas e trocar aquelas que representarem uma melhoria para o tamanho total do ciclo. Todos os pares são escaneados várias vezes, até que uma passada completa não encontre nenhuma melhoria possível. Como o ciclo inicial é produzido com um algorítmo guloso, um resultado conhecido diz que em média são necessárias $O(V)$ passadas pelos pares para não ser possível realizar novas melhorias.

Para trocar as arestas, é necessário usar um algorítmo auxiliar que inverte um subset de uma array:

```
    def swapPath(busLine[], i, j)
        i ← i + 1
        enquanto i menor que j:
            temp ← busLine[i]
            busLine[i] ← busLine[j]
            busline[j] ← temp
            i ← i + 1
            j ← j - 1
```

A complexidade é $O(n)$ (no caso, como se trata de uma array de regiões, temos $O(R)$).

```
def optimizedBusLines(numRegions, busLine, distMatrix,
   totalDist)
```

```
    melhorou ← true
    enquanto melhorou é verdadeiro:
        melhorou ← falso
        para cada i em número de regiões - 1 :
            para cada j partindo de i+1 até o último:
                deltaCost ← Diferenca do custo das
  novas arestas em relacão às arestas antigas
                se o deltaCost for menor do que 0:
                    totalDist ← totalDist + deltaCost
                    swapPath(busLine, i, j)
                    melhorou ← true
```

Complexidade: como explicado, em média $O(R)$ iterações são necesśarias para não haver mais melhorias. Os loops de i e j percorrem por R cada um e, se executar o *while* R vezes, significa que foram feitos swaps para a mesma quantidade de iterações no pior casoo, logo temos $O(R \cdot R \cdot R \cdot R) = O(R^4)$.

No total, temos que duas operações dominam a criação da linha de ônibus: A seleção dos pontos e a melhoria da rota inicial. A complexidade total da tarefa fica $O(R^4) + O(R(V logV + R))$.

6 Tarefa 3: Serviço de rotas entre endereços

É necessário desenvolver uma aplicação com o objetivo de auxiliar a mobilidade urbana da cidade.

Os cidadãos devem informar endereço de origem e destino e a aplicação, considerando partida imediata, deverá ser capaz de retornar a rota para fazer tal viagem em menor tempo utilizando os meios de transportes disponíveis na cidade (ônibus, metrô, táxi e transporte não motorizado. O usuário também pode informar o valor máximo que está disposto a pagar para fazer a viagem.

Para buscar a aresta que corresponde ao endereço passado pelo usuário, utiliza-se a função:

```
def findEdge(cidade, região, rua, número, v1, v2,
  dist_v1, dist_v2)
    v ← -1
    vérticeInicial ← -1
    para cada vértice v na região:
```

```
        arestas ← lista de adjacência de v
        para cada aresta em arestas:
            se a aresta.rua igual à rua:
                vérticeInicial ← aresta.início_da_rua
                finalizar loop

    hashTable visitados com máximo de 20 chaves
    nImoveis ← 0

    se vérticeInicial igual a -1:
        # Não foi possível encontrar a rua especificada
        retornar
    caso contrário:
        arestas ← lista de adjacência de vérticeInicial
        para cada aresta em arestas:
            se aresta.rua diferente de rua ou visitados.get(ponta da aresta) diferente de nulo:
                pular iteracão

            visitados.set(vérticeInicial, true)
            vérticeInicial ← outra ponta da aresta
            arestas = lista de adjacência do vérticeInicial

            se o nImoveis <= número e número <= nImoveis + número de imoveis no segmento:

                v1 ← outra ponta da primeira aresta de arestas
                v2 ← vérticeInicial
                dist_v1 ← (número - nImoveis) * tamanho da primeira aresta de arestas / número de imoveis da primeira aresta de arestas
                dist_v2 ← tamanho da primeira aresta de arestas - dist_v1
                retorna
            nImoveis += número de imoveis da primeira aresta de arestas

    se número > nImoveis:
```

```
        retornar "Não foi possível encontrar"

    retornar
```

Sua complexidade é data por uma cadeia de loops, com compl. $V' + E'$ e outro loop (E') com operações de tempo constante, logo a complexidade é $O(V')$, onde V' e E' são o número de vértices e arestas de uma única região, que por sua vez, é esparsa.

A função $copyStreetInfo$ (Abaixo) procura a aresta desejada e retorna uma cópia da mesma:

```
def copyStreetInfo(cidade, v1, v2)
    aresta ← nova aresta vazia
    arestas ← lista de adjacência de v1

    para todo e em arestas:
        se outra ponta de e igual à v2:
            aresta.maxSpeed ← e.maxSpeed
            aresta.nBuildings ← e.nBuildings
            aresta.nComercial ← e.nComercial
            aresta.nResidential ← e.nResidential
            aresta.nIndustrial ← e.nIndustrial
            aresta.nTouristic ← e.nTouristic
            retornar aresta
    retornar nulo
```

Por percorrer uma única vez em cada aresta fazendo operações de tempo constante, sua complexidade é $O(E)$.

A função $findRoute$ tem como objetivo encontrar uma rota dada as especificações passadas nos parametros.

```
def findRoute(cidade, endereco1[], endereco2[], rota[],
   distãncias[])
    v1, v2 ← -1, -1
    dist_v1, dist_v2 = -1, -1

    findEdge(cidade, endereco1[0], endereco1[1],
   endereco1[2], v1, v2, dist_v1, dist_v2)
```

```
    v3, v4 ← -1, -1
    dist_v3, dist_v4 ← -1, -1

    findEdge(cidade, endereco2[0], endereco2[1], endereco2[2], v3, v4, dist_v3, dist_v4)

    se algum entre v1, v2, v3 e v4 igual à -1:
        retornar "Não foi possível encontrar uma rota"

    edge1 ← copyStreetInfo(cidade, v1, v2)
    edge2 ← copyStreetInfo(cidade, v1, v2)
    edge1.tamanho ← dist_v1
    edge2.tamanho ← dist_v2

    edge3 ← copyStreetInfo(cidade, v3, v4)
    edge4 ← copyStreetInfo(cidade, v3, v4)
    edge3.tamanho ← dist_v3
    edge4.tamanho ← dist_v4

    vTemp1, vTemp2 = Vértices temporários ainda não encontrados

    cidade.addSegment(vTemp1, v1, edge1)
    cidade.addSegment(vTemp1, v2, edge2)
    cidade.addSegment(v3, vTemp2, edge3)
    cidade.addSegment(v4, vTemp2, edge4)

    Dijkstra(grafo, vTemp1, rota, distância)

    cidade.removeSegment(vTemp1, v1)
    cidade.removeSegment(vTemp1, v2)
    cidade.removeSegment(v3, vTemp2)
    cidade.removeSegment(v4, vTemp2)

    retorna true
```

A função invoca as funções $findEdge$ e $copyStreetInfo$ que têm complexidades $O(V')$ e $O(E)$ e também $Dijkstra$ que tem custo $O(VlogV)$, além de operações de complexidade constante. Portanto a complexidadeda função é $O(VlogV)$.

Abaixo estão as funções que pegam a velocidade média dos meios de transporte na aresta selecionada:

```
def compareCar(aresta)
    velocidade = aresta.velocidadeMaxima
    rua = aresta.rua
    região = aresta.região
    velocidade = detTraffic(rua, região, velocidade);
    retorna (node->lenght) / (speed / 3.6)

def compareBus(aresta)
    velocidade = 70
    rua = aresta.rua
    região = aresta.região
    rua = detTraffic(rua, região, velocidade)
    retorna (tamanho da aresta/ (velocidade / 3.6)
}

def compareWalking(aresta)
    retorna (tamanho da aresta / (5 / 3.6)

def compareSubway(aresta)
    retorna (tamanho da aresta) / (70 / 3.6)
```

Essas funções retornam a velocidade média de cada meio de transporte na aresta. As funções fazem apenas operações em $\theta(1)$, logo suas complexidades são também $\theta(1)$.

A função $findClosestSubway$ procura a proximidade entre as estações de metrô, com complexidade $O(R)$ pois passa por cada estação (uma por região).

```
def findClosestSubway(cidade, distancias)
    minDist = INFINITO
    v = -1;
    para cada estacão de metro:
        se distancias[estacão] < minDist:
            minDist = distancias[estacão];
```

```
            v = estacão;
    retorna v
```

A função *findClosestBus* tem o mesmo intuito e formato que a anterior, porém para os pontos de ônibus.

```
def findClosestBus(cidade, distancias)
    minDist = INFINITO
    v = -1;
    para cada ponto de onibus:
        se ditancias[ponto] < minDist:
            minDist = dists[ponto]
            v = ponto
    retorna v;
```

A função abaixo calcula o custo mínimo para ir de um ponto v1 a um ponto v2 do grafo utilizando Dijkstra. Sua complexidade é a mesma do Disjkstra.

```
def minCost(grafo, v1, v2, func)
    parents[grafo.numNodes]
    distance[graph.numNodes]
    grafo.Dijkstra(v1, parents, distance, func)
    return distance[v2]
```

Com ideia semelhante à *findRote*, a *findBestRote* busca as melhores rotas usando a *findRote* e retorna o melhor dos casos. Sua complexidade final é a mesma da *findRote*.

```
def findBestRoute(cidade, metrô, onibus, endereco1,
   endereco2, info, custoMax) {
    v_from = cidade.numNodes - 1
    v_to = cidade.numNodes - 2

    routeForwardWalk[cidade.numNodes];
    routeBackwardWalk[cidade.numNodes];
    distanceForwardWalk[cidade.numNodes];
```

```
    distanceBackwardWalk[cidade.numNodes];

    routeForwardCar[cidade.numNodes];
    routeBackwardCar[cidade.numNodes];
    distanceForwardCar[cidade.numNodes];
    distanceBackwardCar[cidade.numNodes];

    se findRoute(cidade, endereco1, endereco2, routeForwardWalk, distanceForwardWalk, compareWalking) é falso:
        retorna
    se findRoute(cidade, endereco2, endereco1, routeForwardWalk, distanceForwardWalk, compareWalking) é falso:
        retorna

    se findRoute(cidade, endereco1, endereco2, routeForwardCar, distanceForwardCar, compareCar) é falso:
        retorna
    se findRoute(cidade, endereco2, endereco1, routeForwardCar, distanceForwardCar, compareCar) é falso:
        retorna
    vWSF = findClosestSubway(cidade, distanceForwardWalk)
    vWSB = findClosestSubway(cidade, distanceBackwardWalk)
    vWBF = findClosestBus(cidade, distanceForwardWalk)
    vWBB = findClosestBus(cidade, distanceBackwardWalk)

    vCSF = findClosestSubway(cidade, distanceForwardCar)
    vCSB = findClosestSubway(cidade, distanceBackwardCar)
    vCBF = findClosestBus(cidade, distanceForwardCar)
    vCBB = findClosestBus(cidade, distanceBackwardCar)

    times[número de rotas]

    // Walking -> Walking
```

```
    times[0] = distanceForwardWalk[v_to]

    // Car -> Car
    times[1] = distanceForwardCar[v_to]

    // Walking -> Subway -> Walking
    times[2] = distanceForwardWalk[vWSF] +
   minCost(metro, vWSF, vWSB, compareSubway) +
   distanceBackwardWalk[vWSB]

    // Walking -> Bus -> Walking
    times[3] = distanceForwardWalk[vWBF] +
   minCost(onibus, vWBF, vWBB, compareBus) +
   distanceBackwardWalk[vWBB]

    // Car -> Subway -> Car
    times[4] = distanceForwardCar[vCSF] +
   minCost(metrô, vCSF, vCSB, compareSubway) +
   distanceBackwardCar[vCSB]

    // Car -> Bus -> Car
    times[5] = distanceForwardCar[vCBF] +
   minCost(onibus, vCBF, vCBB, compareBus) +
   distanceBackwardCar[vCBB]

    minTime = INFINITO
    bestRoute = -1

    para cada i em número de rotas:
        se times[i] < minTime ou bestRoute == -1:
            minTime = times[i]
            bestRoute = i

    indica qual melhor rota encontrada
```

6.1 API de Trânsito

Para que se possa computar o tempo levado para se locomover de um ponto a outro em uma cidade, além da distância, também é necessário levar em consideração o tráfego de veículos em cada trecho. Com base nisso, foi criada uma API, com o objetivo de, dado o ID de uma rua, sua região e seu limite de velocidade máxima, determinar uma velocidade máxima real para aquele trecho.

Dado que o trabalho trata-se de uma simulação, resolveu-se calcular o trânsito com base na seguinte fórmula:

$$V_t = R_p \cdot M_s \cdot (1 - R_f) \tag{1}$$

Tal que:

- V_t: Velocidade máxima passível de ser alcançada na rua
- R_p: Valor que altera a velocidade máxima com base na seguinte fórmula: $R_p = 0,2 \cdot (region\%3) + 0,8$
- M_s: Velocidade máxima permitida em uma dada rua.
- R_f: Valor dado por: $R_f = (street\%12) \cdot (\frac{T_p}{10})$, com T_p dado por uma função discreta que recebe a hora atual (Inteiro de 0 a 23) e retorna um nível de 0 a 10 de quão alto é o tráfico durante esse horário.

A função $detTraffic$ recebe uma rua especifica e calcula a velocidade na mesma de acordo com o transito no horario real.

```
def detTraffic(street, region, maxSpeed)
    howBusy = street  12regionPenalty = 0.2 * (region now =
  tempo atualtimePenalty = penalidade de tempo de acordo com o
  horárioreductionFactor = (howBusy/12 * (timePenalty/10)
    result = regionPenalty * (maxSpeed * (1 -
  reductionFactor))

    return result
```