Fundação Getúlio Vargas
Escola de Matemática Aplicada

Ciência de Dados e Inteligência Artificial

Projeto e Análise de Algoritmos

Kaiky Eduardo Alves Braga
Larissa Lemos Afonso
Luciano Pereira Sampaio
Samuel Corrêa Lima

Rio de Janeiro
Dezembro / 2024

Sumário

1 Modelagem da Cidade **3**
- 1.1 Estruturas de Dados . . . 3
- 1.2 Funções das Estruturas . . . 4
- 1.3 Geração dos dados . . . 5

2 Tarefa 1 - Linhas de Metrô **7**
- 2.1 Construção das Estações de Metrô . . . 7
- 2.2 Construção das Linhas de Metrô . . . 11

3 Tarefa 2 - Linha de Ônibus **14**
- 3.1 Estações de Ônibus . . . 17
- 3.2 Definição de Rota de Ônibus . . . 17
- 3.3 Pós-Tratamento da Rota . . . 20

4 Tarefa 3 - Serviço de Rotas **20**

5 Tempos de Execução **23**
- 5.1 Tarefa 1 . . . 23
- 5.2 Tarefa 2 . . . 23
- 5.3 Tarefa 3 . . . 24

1 Modelagem da Cidade

Com base no conhecimento geográfico e urbano da cidade, é possível construir uma representação em grafos. Onde, cada cruzamento é representado por um vértice, enquanto as arestas correspondem aos segmentos das ruas que conectam os cruzamentos.

Considerando a locomoção rodoviária, utilizamos um grafo direcionado, em que as arestas indicam os sentidos permitidos para o tráfego de veículos.

Além disso, mantemos uma estrutura complementar que leva em conta o mapa direcionado da cidade. Essa estrutura permite extrair informações tanto sobre a locomoção de pedestres quanto sobre as linhas de metrô, que estão localizadas sob as arestas representadas no grafo.

O grafo foi modelado utilizando uma lista de adjacência. Como cada cruzamento, em média, possui 4 arestas adjacentes (representando conexões entre ruas), e o número de arestas é proporcional ao número de vértices, o grafo é classificado como esparso. Essa característica oferece vantagens, como menor consumo de memória e maior eficiência nas operações subsequentes.

1.1 Estruturas de Dados

Vertex

- `int` **ID**: Identificador do vértice
- `int` **CEP**: CEP onde está situado o vértice

EdgeNode

- `Vertex*` **v1**: Vértice de saída da aresta
- `Vertex*` **otherVertex**: Vértice de chegada da aresta
- `int` **weight**: Distância, em metros, entre os cruzamentos
- `int` **nMin**: Menor número de um imóvel em um trecho
- `int` **nMax**: Maior número de um imóvel em um trecho
- `int` **street**: Número da rua
- `int` **excavationCost**: Custo de escavação de um trecho
- `float` **taxiTime**: Tempo em segundos para percorrer o trecho de táxi
- `float` **walkTime**: Tempo em segundos para percorrer o trecho a pé
- `float` **subTime**: Tempo em segundos para percorrer o trecho com metrô

EdgeNode

- `int` **houseWeight**: Quantidade de imóveis turísticos e comerciais em um trecho
- `int` **traffic**: Quantificador que trânsito do trecho
- `bool` **isBusLine**: Indica se passa a linha de ônibus no trecho

Graph

- `int` **numEdges**: Número total de arestas
- `int` **numVertices**: Número total de vértices
- `class EdgeNode**` **edges**: Lista de adjacência do grafo
- `int` **minimumTaxiCost**: Preço mínimo para uma corrida de táxi
- `int` **taxiRate**: Taxa que converte a distância percorrida no preço a ser pago por uma corrida de táxi, desde que seja maior que `minimumTaxiCost`
- `int` **subwayCost**: Preço da passagem de metrô
- `int` **subwayWaitTime**: Tempo de espera do metrô, em segundos
- `int` **busFee**: Preço da passagem de ônibus
- `int` **busWaitTime**: Tempo de espera do ônibus, em segundos

1.2 Funções das Estruturas

A seguir, serão apresentadas as funções mais importantes de cada estrutura (`addEdge` e `removeEdge`), que são utilizadas com grande recorrência nas demais funções.

Ambas as funções percorrem as arestas adjacentes de `v1`. Assim, no pior caso, terão complexidade $O(E)$, quando todas as arestas forem adjacentes à `v1`.

Algorithm 1 `addEdge(v1, v2, weight, excavationCost, nMin, nMax, street, taxiTime, walkTime, subTime, houseWeight, traffic)`

Entrada:
    `v1, v2, weight, excavationCost, nMin, nMax,`
    `street, taxiTime, walkTime, subTime, houseWeight, traffic`
Efeito: Adiciona uma aresta entre `v1` e `v2` no grafo

$newEdge \leftarrow$ nova instância de `EdgeNode` com os parâmetros fornecidos

```
if m_edges[v1.ID()] = null then
    m_edges[v1.ID()] ← newEdge
else
    edge ← m_edges[v1.ID()]
    while edge.next() ≠ null do
        edge ← edge.next()
    edge.setNext(newEdge)
m_numEdges ← m_numEdges + 1
```

Algorithm 2 `removeEdge(v1, v2)`

```
Entrada:
    v1 (vértice inicial),
    v2 (vértice final da aresta a ser removida)
Efeito: Remove a aresta entre os vértices v1 e v2 no grafo

edge ← m_edges[v1.ID()]
previousEdge ← null
while edge ≠ null do
    if edge.otherVertex() = v2 then
        if previousEdge ≠ null then
            previousEdge.setNext(edge.next())
        else
            m_edges[v1.ID()] ← edge.next()
        Delete edge
        break
    previousEdge ← edge
    edge ← edge.next()
```

1.3 Geração dos dados

A ideia foi criar um grafo que seja baseado em uma estrutura de rede $n \times n$ e que seja fortemente conexo.

Primeiro, o algoritmo cria n^2 vértices, onde n é um parâmetro passado para a função *generateGrid*, representando as posições na grade, cada um identificado por um índice único. Essa etapa tem complexidade $O(gridSize^2)$.

Em seguida, sintetizamos um **ciclo hamiltoniano** para garantir conexidade forte. Esse ciclo conecta todos os vértices em uma sequência circular, garantindo que cada vértice seja acessível de qualquer outro. O laço principal percorre todos os $gridSize^2$ vértices e realiza operações constantes em cada iteração, como calcular o próximo vértice e adicionar uma ou duas arestas ao grafo. Assim, a complexidade dessa etapa é também $O(gridSize^2)$.

Por fim, para aumentar a aleatoriedade de grafos sintetizados ainda mais, criamos uma função que adiciona ainda mais arestas aleatórias. O algoritmo utiliza dois laços aninhados para iterar sobre os vértices, e para cada vértice são realizadas até duas operações de adição de arestas (ou suas reversas). Portanto, a complexidade dessa função também é $O(gridSize^2)$.

Ao total, a complexidade de criar o grafo é $O(gridSize^2)$. Segue o pseudocódigo.

Algorithm 3 `generateGrid(graph, gridSize, numPartitions)`

Entrada:
 `graph` (Grafo a ser criado),
 `gridSize` (Tamanho do grid),
 `numPartitions` (Quantidade de regiões)
Saída:
 `vertices` (Vetor de vértices)

$vertices \leftarrow$ cria vértices para a grade $gridSize \times gridSize$
$rng \leftarrow$ gerador de números aleatórios, $current \leftarrow 1$

addHamiltonianCycle(graph, vertices, gridSize, rng, current)
addGridEdges(graph, vertices, gridSize, rng, current)

Algorithm 4 `addHamiltonianCycle(graph, vertices, gridSize, rng, gridSize)`

Entrada:
 `graph` (grafo),
 `vertices` (vértices do grafo),
 `gridSize` (tamanho da grid),
 `rng` (seed aleatória),
 `current` (contador para o nº de ruas)
Efeito:
 `Adiciona ciclo hamiltoniano para assegurar conectividade forte`

for $i \leftarrow 0$ **to** $gridSize^2 - 1$ **do**
 $next \leftarrow (i+1) \bmod gridSize^2$
 Adiciona aresta $vertices[i] \rightarrow vertices[next]$
 if aleatório(0 ou 1) $= 0$ **then**
  Adiciona aresta reversa $vertices[next] \rightarrow vertices[i]$

Algorithm 5 `addGridEdges(graph, vertices, gridSize, rng, current)`

Entrada:
    `graph` (grafo),
    `vertices` (vértices do grafo),
    `gridSize` (tamanho da grid),
    `rng` (seed aleatória),
    `current` (contador para o nº de ruas)
Efeito:
    `Arestas preservando a grid`

for $i \leftarrow 0$ **to** $gridSize - 1$ **do**
    for $j \leftarrow 0$ **to** $gridSize - 1$ **do**
        $current \leftarrow i \cdot gridSize + j$

        if $j + 1 < gridSize$ **then** ▷ Vizinho à direita
            Adiciona aresta $current \rightarrow current + 1$ ou reversa
        if $i + 1 < gridSize$ **then** ▷ Vizinho abaixo
            Adiciona aresta $current \rightarrow current + gridSize$ ou reversa

2 Tarefa 1 - Linhas de Metrô

O primeiro objetivo foi projetar as linhas de metrô da cidade, seguindo os critérios:

- As linhas de metrô serão escavadas abaixo das ruas da cidade.
- As estações estarão localizadas nos cruzamentos (vértices).
- Cada região (identificada pelo CEP) deve possuir uma estação de metrô.
- A localização de cada estação deve minimizar a distância entre ela e o ponto mais distante da sua respectiva região.
- A planta da cidade atribui a cada segmento de rua um custo de escavação, que representa o investimento necessário para integrá-lo à linha de metrô.

2.1 Construção das Estações de Metrô

Para determinar os vértices onde as estações seriam construídas, foi implementada uma função específica. Essa função recebe como entrada um CEP da estrutura da cidade e retorna um subgrafo que contém:

- Todos os vértices associados ao CEP fornecido;
- Todas as arestas cujos dois vértices também pertencem ao mesmo CEP.

A operação possui um custo de $O(V+E)$, pois percorre toda a lista de adjacência para identificar os elementos que atendem a essa condição.

Algorithm 6 `Graph::getVertexIDsByCEP(cep, vertices)`

Entrada: `cep` (CEP desejado), `vertices` (lista de vértices)
Saída: Lista de IDs dos vértices associados ao CEP fornecido

$ids \leftarrow$ lista vazia
for all $vertex \in vertices$ **do**
    if $vertex.CEP() = cep$ **then**
        $ids \leftarrow ids \cup vertex.ID()$
return ids

Algorithm 7 `Graph::generateSubgraphByCEP(cep, vertices)`

Entrada: `cep` (CEP desejado), `vertices` (lista de vértices)
Saída: Subgrafo contendo vértices e arestas associados ao CEP fornecido

$subgraph \leftarrow$ novo grafo com tamanho $|vertices\_cidade|$
for all $v1 \in vertices$ **do**
    if $v1.CEP() \neq cep$ **then**
        continue
    while $edge \neq$ `null` **do**
        $v2 \leftarrow edge.otherVertex()$
        if $v2.CEP() = cep$ **then**
            $subgraph.addEdge(v1, v2)$
return $subgraph$

Em seguida, foi implementada uma função para gerar a árvore CPT utilizando o algoritmo de Dijkstra, adaptado para calcular as distâncias dos vértices a um determinado nó. Optamos por sua versão com *heap* de prioridade, que apresenta uma complexidade de $O((V+E)\log V)$. Considerando que o grafo é esparso, essa complexidade se reduz, na prática, a $O(V \log V)$. Além disso, dentro da função é possível fazer comparações de diferentes métricas de distâncias através do parâmetro `usedWeight`, permitindo maior versatilidade entre as demais problemáticas do projeto.

Algorithm 8 `Graph::cptDijkstra(v0, parent, dist, usedWeight)`

```
Entrada:
    v0 (Vértice de origem),
    parent (Vetor de pais),
    dist (Vetor de distâncias),
    usedWeight (Tipo de Métrica de distância)
Saída: Vetores parent e dist atualizados

Inicialização:
for v = 0 to |V| − 1 do
    parent[v] ← −1
    dist[v] ← ∞
    checked[v] ← false
parent[v0.ID()] ← v0.ID()
dist[v0.ID()] ← 0
heap ← prioridade fila de pares (dist[v0.ID()], v0.ID())
while a fila de prioridade não estiver vazia do
    v1 ← vértice com a menor distância em heap
    heap.pop()
    if dist[v1] = ∞ then
        break
    if checked[v1] then
        continue
    checked[v1] ← true
    edge ← lista de arestas do vértice v1
    while edge ≠ null do
        v2 ← edge.otherVertex()
        v2ID ← v2.ID()
        if checked[v2ID] = false then
            cost ← edge.getParam(usedWeight)
            if dist[v1] + cost < dist[v2ID] then
                dist[v2ID] ← dist[v1] + cost
                parent[v2ID] ← v1
                heap.push((dist[v2ID], v2ID))
        edge ← edge.next()
```

Após isso, foi implementada a função responsável por determinar o vértice ideal para a construção de uma estação de metrô. O processo começa executando a função que identifica o subgrafo correspondente ao CEP especificado. Em seguida, o algoritmo de Dijkstra é aplicado a partir de cada vértice do subgrafo. Durante essa etapa, para cada vértice analisado, é calculada a maior distância entre ele e os demais vértices do subgrafo. O vértice que apresentar a menor dessas distâncias máximas é escolhido como o local ideal para a construção da estação.

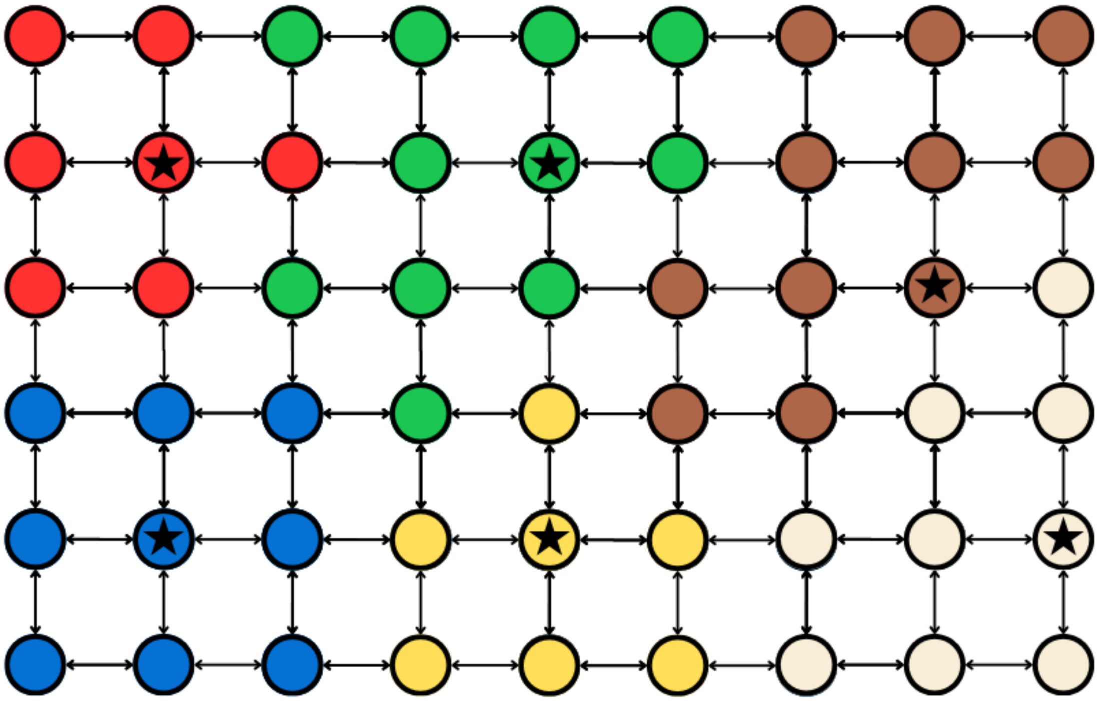
A complexidade total desse processo é $O(V + E) + O(V^2 \log V)$, o que resulta em $O(V^2 \log V)$, devido à execução do algoritmo de Dijkstra para todos os vértices do subgrafo. Assim, a complexidade para achar todos os vértices das estações para cada CEP se torna $O(|CEP|V^2 \log V)$.

Algorithm 9 `findOptimalVertex(cep, vertices, numVertices)`

```
Entrada: cep (CEP desejado), vertices (lista de vértices), numVertices
(número total de vértices)
Saída: Vértice ótimo que minimiza a maior distância no grafo

subgraph ← generateSubgraphByCEP(cep, vertices)
vertexIDs ← subgraph.getVertexIDsByCEP(cep, vertices)
if vertexIDs está vazio then
    Lançar exceção: "Nenhum vértice encontrado para o CEP fornecido."
minMaxDistance ← INT_MAX
optimalVertex ← null
for all vertexID ∈ vertexIDs do
    parent[numVertices] ← vetor de inteiros
    dist[numVertices] ← vetor de inteiros
    subgraph.cptDijkstra(vertices[vertexID], parent, dist,
"weight")
    maxDistance ← 0
    for int i = 0 to numVertices - 1 do
        if dist[i] ≠ INT_MAX then
            maxDistance ← max(maxDistance, dist[i])
    if maxDistance ¡ minMaxDistance then
        minMaxDistance ← maxDistance
        optimalVertex ← vertices[vertexID]
return optimalVertex
```

A seguinte ilustração mostra como ficaria o grafo não direcionado da cidade após achar todos os vértices ótimos das estações de metrô. Cada cor representa um CEP distinto, e as estrelas apontam a localização das estações de metrô.



2.2 Construção das Linhas de Metrô

Para construir as linhas de metrô, foi necessário definir os segmentos a serem escavados de forma que o custo total para a cidade fosse minimizado. Para esse propósito, foi desenvolvida a função **mstSubway()**, que gera uma Árvore Geradora Mínima (MST) para as linhas de metrô. Temos que: cada vétice representa uma estação e cada aresta representa o menor custo entre as estações.

A função começa executando o algoritmo de Dijkstra para cada estação, a fim de determinar os menores custos de escavação entre cada estação e todos os demais vértices do grafo original. Como o Dijkstra utilizado emprega um heap, sua complexidade é $O(|CEP|(V+E)logV)$, onde CEP representa o número de estações (temos uma estação por CEP), enquanto V é o número de vértices e E, o número de arestas do grafo. Como o grafo original é esparso, isso atinge complexidade $O(|CEP|VlogV)$

Em seguida, é construído um novo grafo completo onde os vértices correspondem às estações, e as arestas representam os custos de escavação entre elas. A construção desse grafo tem complexidade $O(|CEP|^2)$.

Por fim, utiliza-se o algoritmo de Prim, adequado para grafos densos, com complexidade $O(|CEP|^2)$, para encontrar a MST do grafo de estações. Como resultado, obtemos um plano para as linhas de metrô com custo mínimo de construção em $O(|CEP|VlogV)$ pois $|CEP|$ não é proporcional a V e $|CEP| < V$.

Algorithm 10 `mstSubway(graph, numVertices, stationsList, numStations, pathEdges)`

```
Entrada:
    graph (grafo),
    numVertices (número total de vértices),
    stationsList (lista de estações),
    numStations (número de estações),
    pathEdges (arestas da MST das linhas de metrô)
Saída: Lista de arestas da MST das linhas de metrô
parentList[numStations][numVertices] ← matriz de inteiros
costMatrix[numStations][numVertices] ← matriz de inteiros
for v = 0 to numStations - 1 do
    graph.cptDijkstra(graph, stationsList[v], parentList[v], costMatrix[v], "excavationCost")
stationsVertices ← lista vazia
for i = 0 to numStations - 1 do
    stationVertex ← novo vértice com ID i e CEP(stationsList[i])
    stationsVertices.push_back(stationVertex)
g ← novo grafo completo com numStations vértices
for i = 0 to numStations - 1 do
    for j = i + 1 to numStations - 1 do
        station1 ← stationsList[i]
        station2 ← stationsList[j]
        if costMatrix[i][station2.ID] < costMatrix[j][station1.ID] then
            minCost ← costMatrix[i][station2.ID]
        else
            minCost ← costMatrix[j][station1.ID]
        g.addEdge(stationsVertices[i], stationsVertices[j], minCost)
        g.addEdge(stationsVertices[j], stationsVertices[i], minCost)
mstParentList[numStations] ← vetor de inteiros
g.mstPrimFastV1(mstParentList, numStations, "excavationCost")
mstEdges ← lista vazia
g.edgesFromParent(mstParentList, numStations, mstEdges)
for cada mstEdge em mstEdges do
    v1i ← mstEdge.v1.ID
    v2i ← mstEdge.otherVertex.ID
    v1 ← stationsList[v1i].ID
    v2 ← stationsList[v2i].ID
    graphEdges ← lista vazia
    graph.pathFromParent(v2, parentList[v1i], graphEdges)
    distance ← soma dos pesos das arestas em graphEdges
    pathEdges.push_back((v1, v2, distance))
    graphEdges.clear()
```

Algorithm 11 `mstPrimFastV1(parent, numVertices, usedWeight)`

```
Entrada: parent (vetor para armazenar os pais na MST), numVertices (número de vértices), usedWeight (peso a ser usado nas arestas)
Saída: Vetor parent preenchido com os pais na MST

if numVertices ≠ 0 then
    m_numVertices ← numVertices
checked[m_numVertices] ← vetor de booleanos inicializado como false
vertexCost[m_numVertices] ← vetor de inteiros inicializado como INT_MAX
for v = 0 to m_numVertices - 1 do
    parent[v] ← −1
    checked[v] ← false
    vertexCost[v] ← INT_MAX
parent[0] ← 0
checked[0] ← true
edge ← lista de arestas adjacentes ao vértice 0
while edge não for null do
    v2 ← outro vértice da aresta edge
    v2ID ← ID de v2
    parent[v2ID] ← 0
    vertexCost[v2ID] ← edge.getParam(usedWeight)
    edge ← próxima aresta em m_edges[0]
while true do
    minCost ← INT_MAX
    v1 ← null
    for v = 0 to m_numVertices - 1 do
        if !checked[v] and vertexCost[v] < minCost then
            minCost ← vertexCost[v]
            v1 ← v
    if minCost == INT_MAX then
        break
    checked[v1] ← true
    edge ← lista de arestas adjacentes a v1
    while edge não for null do
        v2 ← outro vértice da aresta edge
        v2ID ← ID de v2
        cost ← edge.getParam(usedWeight)
        if !checked[v2ID] and cost < vertexCost[v2ID] then
            vertexCost[v2ID] ← cost
            parent[v2ID] ← v1
        edge ← próxima aresta em m_edges[v1]
```

3 Tarefa 2 - Linha de Ônibus

A segunda tarefa consiste em determinar uma linha de ônibus *hop-on/hop-off* que percorre todas as regiões da cidade. A rota definida deve obrigatoriamente formar um ciclo, iniciando e terminando no mesmo local, garantindo que ao menos um vértice de cada região esteja incluído no trajeto.

Além disso, buscou-se otimizar o trajeto considerando dois objetivos principais: **maximizar** o número de imóveis comerciais e atrações turísticas ao longo da rota e **minimizar** a quantidade de imóveis residenciais e industriais. Para isso, foi desenvolvida uma heurística baseada em pesos atribuídos às arestas do grafo, definida pela seguinte fórmula:

$$\textbf{houseWeight} = \frac{\text{N}^{\text{o}} \text{ de imóveis residenciais}}{\text{N}^{\text{o}} \text{ de imóveis comerciais} + \text{N}^{\text{o}} \text{ de imóveis residenciais}}$$

Para definir a rota do ônibus, foi adotada a hipótese de que os vértices fronteiriços obrigatoriamente fariam parte do ciclo. Consideraram-se como vértices fronteiriços aqueles conectados por arestas a vértices associados a diferentes CEPs. A identificação desses vértices foi organizada em uma tabela hash estruturada da seguinte forma:

$$\textbf{HashMap} = \{\text{CEP : vetor de IDs de vértices fronteiriços associados a esse CEP}\}$$

Algorithm 12 Graph::findBorder(vertices, ceps)

Entrada:
  `vertices` (lista de vértices do grafo)
  `ceps` (lista de CEPs associados aos vértices)
Saída:
  `hashMap` (estrutura que mapeia cada CEP para os IDs dos seus vértices de borda)

Inicialização:
  `hashMap` $\leftarrow$ estrutura vazia
for all `cep` em `ceps` **do**
  `hashMap[cep]` $\leftarrow$ vetor vazio
Identificação de vértices de borda:
for $i = 0$ **to** $|V| - 1$ **do**
  $v1 \leftarrow vertices[i]$
  $edge \leftarrow$ lista de arestas conectadas a $v1$
  $v1CEP \leftarrow v1.CEP()$
  $v1ID \leftarrow v1.ID()$
  while $edge \neq$ `null` **do**
    $v2 \leftarrow edge.otherVertex()$
    $v2CEP \leftarrow v2.CEP()$
    $v2ID \leftarrow v2.ID()$
    if $v1CEP \neq v2CEP$ **then**
      `hashMap[v1CEP].insert(v1ID)`
      `hashMap[v2CEP].insert(v2ID)`
    $edge \leftarrow edge.$`next()`
Remoção de duplicatas nos vértices de borda:
for all `cep` em `ceps` **do**
  `vectorCEP` $\leftarrow$ `hashMap[cep]`
  `uniqueVertices` $\leftarrow$ conjunto único de `vectorCEP`
  `hashMap[cep]` $\leftarrow$ vetor de `uniqueVertices`
return `hashMap`

Na primeira etapa, ocorre a inicialização do `hashMap`, onde, para cada CEP presente no vetor `ceps`, é criada uma entrada no `hashMap` com uma lista vazia. Como esta etapa percorre linearmente o vetor `ceps`, sua complexidade é $O(C)$, onde C é o número total de CEPs.

Na segunda etapa, o algoritmo percorre todos os vértices do grafo (`pela lista vertices`) e, para cada vértice, todas as suas arestas adjacentes. Durante essa iteração, ele compara os CEPs dos vértices conectados pelas arestas e atualiza o `hashMap` se os vértices pertencerem a CEPs diferentes, indicando que são vértices de borda. Como cada vértice é processado uma vez e cada aresta é analisada no máximo duas vezes (uma para cada vértice conectado), a complexidade dessa etapa é $O(V + E)$, onde V é o número de vértices e E é o número de arestas no grafo.

Na terceira etapa, o algoritmo remove duplicatas da lista de vértices associada a cada CEP no `hashMap`. Para cada CEP, a lista de vértices é convertida em um conjunto (`set`) para eliminar duplicatas, e depois reconvertida para uma lista. Se o número médio de vértices associados a cada CEP for $\texttt{avg}(N)$, a complexidade dessa etapa é $O(C \cdot \texttt{avg}(N))$. No pior caso, todos os vértices podem ser marcados como borda para o mesmo CEP, o que resultaria em $\texttt{avg}(N) = V$, levando a uma complexidade de $O(C \cdot V)$.

Ao combinar essas etapas, a complexidade total do algoritmo é $O(C) + O(V + E) + O(C \cdot \texttt{avg}(N))$. No pior caso, quando muitos vértices de borda estão associados a cada CEP, a complexidade final é $O(V + E + C \cdot V)$. No entanto, se o número de CEPs (C) for pequeno em relação a V, a complexidade dominante será $O(V + E)$.

Em seguida, implementou-se a função de geração de IDs por CEP, que atuará como uma função auxiliar em etapas futuras.

Algorithm 13 `generateIndexCEPS(ceps, numCEPS)`

Entrada: `ceps` (array de valores dos CEPs), `numCEPS` (tamanho do array)
Saída: `cepOrderMap` (mapa não ordenado que associa cada CEP ao seu índice ordenado)

$cepsWithIndices \leftarrow$ array de pares (valor do CEP, índice original)
for $i \leftarrow 0$ até $numCEPS - 1$ **do**
    $cepsWithIndices[i] \leftarrow (ceps[i], i)$

`sort(`$cepsWithIndices$`, `$cepsWithIndices + numCEPS$`)`
$cepOrderMap \leftarrow$ nova tabela hash
for $i \leftarrow 0$ até $numCEPS - 1$ **do**
    $cepOrderMap[cepsWithIndices[i].first] \leftarrow i$

return $cepOrderMap$

Dado um array de CEPs, primeiro realizamos sua ordenação, que possui complexidade de $O(|CEP| \log |CEP|)$. Após a ordenação, construímos uma HashTable da seguinte forma:

$$\textbf{cepOrderMap} = \{\text{CEP} : \text{ID\_CEP}\}$$

Aqui, ID_CEP representa a posição do CEP no array ordenado. Essa operação é realizada para todos os elementos do array, o que possui complexidade $O(|CEP|)$. Assim, a complexidade dominante do algoritmo é $O(|CEP| \log |CEP|)$, devido à etapa de ordenação.

3.1 Estações de Ônibus

A seguir, foi implementada a função responsável pela seleção das estações de ônibus, na qual um nó fronteira é escolhido aleatoriamente para cada CEP. A estação de ônibus associada a cada CEP foi selecionada de forma aleatória a partir de um conjunto de vértices borda. A complexidade dessa operação é $O(|CEP|)$, onde $|CEP|$ representa o número de CEPs envolvidos no processo.

Algorithm 14 `createBusStations(verticesBorda, ceps, stations, numCEPS)`

Entrada: `verticesBorda` (mapeamento de CEPs para vértices borda),
  `ceps` (lista de CEPs), `stations` (lista de estações de ônibus), `numCEPS`
  (número de CEPs)
Saída: `stations` preenchido com vértices aleatórios das bordas para cada CEP

for $i \leftarrow 0$ até $numCEPS - 1$ **do**
  $vertices \leftarrow verticesBorda.search(ceps[i])$
  $sizeVerticesBorda \leftarrow |vertices|$
  $randomIndex \leftarrow$ número aleatório entre 0 e $sizeVerticesBorda - 1$
  $stations[i] \leftarrow vertices[randomIndex]$

3.2 Definição de Rota de Ônibus

A função que cria o ciclo de menor custo a partir de uma matriz de distâncias entre CEPs possui uma complexidade de $O(|CEP|!)$. Já que o algoritmo testa todas as permutações possíveis dos CEPs para encontrar o ciclo de menor custo, o que resulta em um total de $|CEP|!$ permutações. Para cada permutação, é necessário calcular o custo do ciclo, o que envolve iterar sobre todos os CEPs e somar as distâncias correspondentes, o que tem complexidade $O(|CEP|)$. Assim, a complexidade total da função é $O(|CEP|! \cdot |CEP|)$, onde o fator dominante é o número de permutações $|CEP|!$.

Algorithm 15 `createCicle(sizeMatrix, matrixDist)`

Entrada: `sizeMatrix` (tamanho da matriz de distâncias), `matrixDist` (matriz de distâncias entre os vértices)
Saída: O ciclo de menor custo e seu valor total

$vertCEPS \leftarrow$ vetor contendo os índices de 0 até $sizeMatrix - 1$
for $i \leftarrow 0$ até $sizeMatrix - 1$ **do**
    $vertCEPS[i] \leftarrow i$
$minCost \leftarrow \infty$
$bestPath \leftarrow$ lista vazia
repeat
    $currentCost \leftarrow 0$
    for $i \leftarrow 0$ até $sizeMatrix - 2$ **do**
        $currentCost \leftarrow currentCost + matrixDist[vertCEPS[i]][vertCEPS[i+1]]$
    $currentCost \leftarrow currentCost + matrixDist[vertCEPS[sizeMatrix-1]][vertCEPS[0]]$
    if $currentCost < minCost$ **then**
        $minCost \leftarrow currentCost$
        $bestPath \leftarrow vertCEPS$
until não houver mais permutações de $vertCEPS$
$bestPath.push\_back(bestPath[0])$
return $(bestPath, minCost)$

A função `findBusLine` possui uma complexidade de $O(|CEP| \cdot (V \log V) + |CEP|!)$, onde $|CEP|$ é o número de CEPs e V é o número de vértices no grafo. A função começa com dois laços aninhados: o primeiro, de $O(|CEP|)$, inicializa um vetor de uso dos vértices, e o segundo, de $O(|CEP|)$, calcula a matriz de distâncias entre os CEPs utilizando o algoritmo de Dijkstra, que tem complexidade $O(V \log V)$ para cada execução. Assim, essa parte da função tem complexidade total de $O(|CEP| \cdot (V \log V))$. A seguir, a função `createCicle` é chamada para calcular o ciclo de menor custo, e essa etapa tem complexidade $O(|CEP|!)$, devido ao uso de permutações para testar todas as combinações possíveis dos CEPs. Por fim, há um laço de $O(|CEP|)$ para reconstruir a rota de ônibus, onde o Dijkstra é novamente usado para cada trecho do ciclo, o que adiciona uma complexidade de $O(|CEP| \cdot (V \log V))$. Portanto, a complexidade total da função `findBusLine` é dominada por $O(|CEP| \cdot (V \log V) + |CEP|!)$, sendo que $|CEP|!$.

Algorithm 16 `findBusLine(graph, verticesCEPS, vertices, refCEPs, matrixWeightCEPS)`

```
Entrada: graph (grafo), verticesCEPS (vetor de vértices que representam os
    CEPs ordenados), vertices (vetor de todos os vértices), refCEPs (mapa de
    referência dos CEPs), matrixWeightCEPS (matriz de pesos entre os CEPs)
Saída: Rota de ônibus representada pelos vértices no ciclo

numCEPS ← tamanho de verticesCEPS
numVert ← tamanho de vertices
inUse ← vetor de tamanho numVert inicializado como false

for i ← 0 até numCEPS − 1 do
    inUse[verticesCEPS[i].ID()] ← true
for i ← 0 até numCEPS − 1 do
    cptDijkstra(verticesCEPS[i], parents, dist, "houseWeight")
    for j ← 0 até numVert − 1 do
        if inUse[j] then
            matrixWeightCEPS[i][refCEPs[vertices[j].CEP()]] ← dist[j]
seqNodes ← createCicle(numCEPS, matrixWeightCEPS)
seqVector ← seqNodes.first
sizeSeq ← tamanho de seqVector
cicle ← vetor vazio
for i ← 0 até sizeSeq − 1 do
    cicle.push_back(verticesCEPS[seqVector[i]])
result ← vetor vazio
for i ← 0 até sizeSeq − 2 do
    cptDijkstra(cicle[i], parents, dist, "houseWeight")
    current ← cicle[i + 1].ID()
    path ← vetor vazio
    while current ≠ parents[current] do
        path.push_back(parents[current])
        current ← parents[current]
    result.insert(result.end(), path.rbegin(), path.rend())
result.push_back(cicle[0].ID())
return result
```

3.3 Pós-Tratamento da Rota

Após localizar o vetor da rota, atualizamos as arestas com a indicação da variável isBusLine. Para isso, realizamos uma varredura completa no grafo, identificando cada par de vértices consecutivos no ciclo. Em seguida, selecionamos as arestas correspondentes e as marcamos como pertencentes à trajetória do ônibus. Como varremos o grafo todo no pior caso, temos a complexidade de $O(V + E)$.

Algorithm 17 `setBusLine(seqBusLine, vertices)`

```
Entrada:
    seqBusLine (Sequência de IDs de vértices da linha de ônibus)
    vertices (Vetor de vértices do grafo)
Efeito: Marca as arestas pertencentes à linha de ônibus como tal

for i ← 0 até |seqBusLine| − 2 do
    edge ← m_edges[seqBusLine[i]]
    while edge ≠ null do
        if edge.otherVertex().ID() = seqBusLine[i + 1] then
            edge.setIsBusLine(true)
            break
        edge ← edge.next()
```

4 Tarefa 3 - Serviço de Rotas

O objetivo desta parte do trabalho é encontrar o caminho mais rápido em um grafo ponderado, respeitando uma restrição de orçamento P. Para isso usamos a função `findFastestPathWithinBudget`, que utiliza uma abordagem similar ao algoritmo de Dijkstra para minimizar o tempo de deslocamento, considerando diferentes meios de transporte, cada um com custos e tempos distintos.

O algoritmo começa inicializando uma matriz `dist` de dimensões $|V| \times (P + 1)$, onde $|V|$ é o número de vértices no grafo. Essa matriz armazena o menor tempo necessário para alcançar um vértice u com um custo acumulado c. Inicialmente, todos os valores de `dist` são configurados como ∞, exceto `dist[start.ID()][0]`, que é definido como 0, já que o vértice inicial não tem custo ou tempo acumulado. Além disso, é criada uma matriz `parent` que registra o vértice anterior e o custo associado para reconstruir o caminho ao final.

O algoritmo utiliza uma fila de prioridade `queue` para armazenar estados na forma $(tempo, (custo, noOnibus, vértice))$. A fila é ordenada pelo tempo crescente, de forma que os estados com menor tempo acumulado são processados primeiro. A busca começa inserindo o vértice inicial na fila com 0 de tempo e custo.

A cada iteração, o algoritmo processa o estado de menor tempo retirado da fila. Para o vértice atual u, ele examina todas as arestas conectadas a u. Para cada aresta, calcula-se o tempo e o custo necessário para alcançar o vértice adjacente v, considerando os diferentes meios de transporte disponíveis: táxi, ônibus, metrô ou caminhada. O custo e o tempo dependem de variáveis como distância, tráfego e, no caso do ônibus, a taxa de embarque. Caso o custo acumulado seja menor ou igual a P e o tempo acumulado seja menor do que o valor armazenado em `dist[v][newCost]`, o estado é atualizado, e o novo estado é adicionado à fila.

Se o vértice de destino `end` for alcançado, o algoritmo reconstrói o caminho percorrido utilizando a matriz `parent`. O caminho é rastreado do destino para a origem, armazenando os vértices visitados e invertendo a ordem ao final. Caso o vértice `end` não possa ser alcançado dentro do orçamento P, o algoritmo retorna $(-1, \{\})$.

A complexidade do algoritmo depende do número de vértices $|V|$, do número de arestas $|E|$ e do orçamento P. A inicialização das matrizes `dist` e `parent` requer $O(|V|P)$. Durante a execução, cada estado na fila de prioridade é processado no máximo $P+1$ vezes para cada vértice, pois há $P+1$ possíveis valores de custo. Como cada vértice pode ter até $|E|$ arestas incidentes, o número total de operações na fila de prioridade é $O(|E|P)$.

A inserção e remoção na fila de prioridade têm custo $O(\log(|V|P))$ devido ao uso de uma heap binária. Portanto, a complexidade total do algoritmo é $O((|E| + |V|)P\log(|V|P))$.

Segue o pseudocódigo.

Algorithm 18 findFastestPathWithinBudget(start, end, P, stations, numPartitions)

Entrada:
  `start` (Vértice de partida)
  `end` (Vértice de chegada)
  `P` (Restrição do preço)

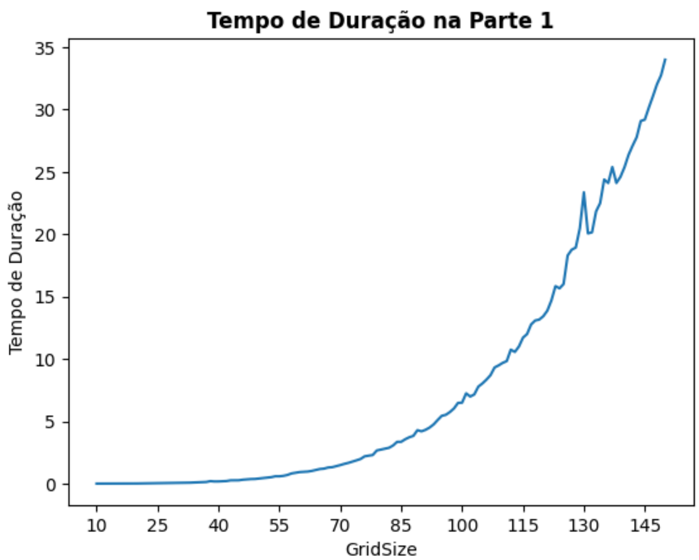
Inicialização:
  `dist`$[u][custo] \leftarrow \infty$ ▷ Menor tempo para chegar em u com $custo$
  `parent`$[u][custo] \leftarrow (-1, -1)$
  `parent[start.ID()][0]` $\leftarrow (-1, start.ID())$
  `dist[start.ID()][0]` $\leftarrow 0$
  `queue.push((0, (0, false, start.ID())))`
  `twoWayCopy` $\leftarrow$ `Graph.copy(twoWay=true)`

```
while !queue.empty() do
    (currentTime, (currentCost, inBus, u)) ← pq.top()
    queue.pop()
    if u = end.ID() then
        path ← lista vazia
        costUsed ← currentCost
        while u ≠ −1 do
            path.push_back(u)
            (prevVertex, prevCost) ← parent[u][costUsed]
            u ← prevVertex, costUsed ← prevCost
        path.reverse()
        return (currentTime, path)
    for twoWayEdge conectado a u do
        v ← edge.otherVertex().ID()
        distance ← edge.weight()
        traffic ← edge.traffic()
        if edge.isBusLine() then
            if inBus then
                busPrice ← 0
            else
                busPrice ← busFee
        else
            busPrice ← INT_MAX
        additionalBusTime ← inBus ?  0 :  busWaitTime
        times ← [max(0, edge.taxiTime + traffic),
                 max(0, edge.taxiTime + traffic + additionalBusTime),
                 edge.subTime, edge.walkTime]
        costs ← [max(minimumTaxiCost, taxiRate · distance),
 busPrice, subwayCost, 0]
        for 0 ≤ i ≤ 3 do
            if costs[i] = ∞ ou times[i] = ∞ then
                continue
            newCost ← currentCost + costs[i]
            newTime ← currentTime + times[i]
            isOneWay ← false
            for oneWayEdge de u do
                if oneWayEdge.otherVertex().ID() == v then
                    isOneWay ← true
            if oneWayEdge OR (i > 1) then
                if newCost ≤ P e newTime < dist[v][newCost] then
                    dist[v][newCost] ← newTime
                    parent[v][newCost] ← (u, currentCost)
                    queue.push((newTime, (newCost, (i = 1), v)))
return (-1, {})
```

5 Tempos de Execução

5.1 Tarefa 1

Anotamos os tempos de execução, em segundos, para a função que cria a malha do metrô em grafos de tamanho $gridSize \times gridSize$, com $gridSize$ variando de 10 a 145 e com 10 regiões. Levando-se em consideração os algoritmos utilizados para a Tarefa 1, a maior complexidade obtida foi $O(|CEP|V^2 logV)$.



5.2 Tarefa 2

Em seguida, avaliamos o tempo das funções usadas para definir as rotas de ônibus em grafos de tamanho $gridSize \times gridSize$ e com $gridSize$ regiões, que varia de 2 a 13. Levando-se em consideração os algoritmos utilizados para a Tarefa 2, observamos que a complexidade é $O(|CEP|(VlogV) + |CEP|!)$. Foram realizados testes com gridSize's pequenos porque a complexidade do $|CEP|!$ é dominante, como pode ser observado na última iteração.

5.3 Tarefa 3

Por fim, avaliamos o tempo da função que determina o caminho mais rápido entre dois pontos dada uma restrição orçamentária (P). Testamos para grafos de tamanho $gridSize \times gridSize$ e com 10 regiões. Nesse caso, a complexidade é $O((|E| + |V|)P \log(|V|P))$.