Projeto planejamento de transporte

Alessandra Belló Soares,
Daniel de Miranda Almeida,
Gustavo Tironi,
João Felipe Vilas Boas
Luís Felipe de Abreu Marciano,
Paloma Vieira Borges

Dezembro, 2024

1 Definições iniciais

Começaremos este relatório destacando algumas configurações do problema, parâmetros predefinidos e peculiaridades na modelagem da cidade. Em seguida, apresentaremos a estrutura inicial em grafo, com seus respectivos pesos e a modelagem adotada. Após isso, definiremos as ideias iniciais que orientaram as implementações das soluções. Por fim, comentaremos algumas decisões de desenvolvimento, visando proporcionar uma melhor compreensão do projeto.

Decisões de Projeto

A modelagem apresentada neste relatório foi desenvolvida especificamente para a cidade de **Vargas**. Dessa forma, as estruturas e algoritmos foram projetados de maneira otimizada, considerando as peculiaridades da cidade em termos de tempo de execução, uso eficiente de espaço e simplicidade na implementação.

Para contextualizar o desenvolvimento, é importante destacar algumas características definidas pela prefeitura que foram fundamentais para a modelagem do algoritmo. Também, devemos nos atentar para a história dessa cidade. Fundada em 1954, ela foi totalmente planejada. Por isso, desde então, algumas regras devem ser seguidas para manter esse planejamento. São elas:

1. **Numeração dos cruzamentos:** Todos os cruzamentos da cidade são numerados. Essa medida foi implementada para facilitar a localização de estabelecimentos situados nas esquinas. Por exemplo, é possível dizer:

   > "Minha loja fica na décima esquina."

   e todos saberão onde encontrá-la (ou pelo menos, essa era a expectativa quando o sistema foi implementado).
2. **Tamanho dos lotes:** Visando promover igualdade, a prefeitura estabeleceu um número fixo de lotes para cada rua da cidade. Com isso, o comprimento total de cada rua é dividido de maneira uniforme entre os lotes, garantindo que todos possuam o mesmo tamanho.
3. **Extensão limitada das ruas:** Cada rua se estende exclusivamente de uma esquina a outra, evitando que ruas ultrapassem mais de um cruzamento.

Apesar de serem nomeadas, recentemente o novo prefeito decidiu que as ruas passariam a ser identificadas pelos cruzamentos que conectam, sempre ordenados do menor para o maior. Embora possa parecer estranho à primeira vista, o sistema funciona da seguinte forma:

Na cidade de Vargas, por exemplo, a **Rua Getúlio** conecta as esquinas 10 e 100. Com a nova nomenclatura, essa rua passa a ser identificada como **(10, 100)**. A ordem dos números segue uma lógica fixa:

o número da menor esquina sempre aparece primeiro, independentemente da direção.

Além disso, o prefeito resolveu reorganizar a numeração dos lotes para corrigir uma desordem existente. Sob a nova regra, ao seguir no sentido da menor esquina para a maior, os lotes do lado direito da rua recebem números pares, começando em 2 e sendo incrementados de 2 em 2 (2, 4, 6, 8, 10...). No lado esquerdo, os lotes são numerados com números ímpares, também em sequência. Essa abordagem visa padronizar e simplificar a identificação dos lotes na cidade.

Estruturas de dados

Recebemos da prefeitura um mapa da cidade, com o qual construímos um grafo onde cada vértice representa um cruzamento e cada aresta representa uma rua. Para armazenar esse grafo, utilizamos a **lista de adjacência**, que nos permite não guardar as informações sobre os cruzamentos, pois estes não contêm dados relevantes para o problema, mas sim apenas as ruas, que possuem as informações que interessam.

Dentro da lista de adjacência, para cada rua (aresta), armazenamos as seguintes informações:

- O número do cruzamento de origem e o número do cruzamento de destino;
- O bairro ao qual a rua pertence;
- O comprimento da rua (quantos metros há entre uma esquina e outra);
- O custo para que ela seja escavada;
- A velocidade máxima permitida na rua;
- Se a rua é de mão única ou mão dupla;
- O número de lotes em cada lado da rua;
- Um array com o número de lotes de cada categoria naquela rua;

```
int lotes[4] = {#casas, #industrias, #atrações, #comercios};
```

Além disso, para aprimorar nossa solução de rotas, decidimos incluir um atributo extra e dinâmico que armazena, e atualiza conforme necessário, as informações de tráfego fornecidas pela API, disponibilizada gentilmente pela prefeitura, que monitora as ruas da cidade.

Desenho da solução

Com todas essas informações em mãos, finalmente conseguimos começar a pensar nas funcionalidades das soluções que vamos criar. Para facilitar, vamos dividir em duas frentes principais:

- **Cumprir com a licitação da prefeitura:**
  - Planejamento das estações de metrô.
  - Planejamento da construção subterrânea do metrô.
  - Planejamento das linhas de ônibus de forma eficiente.
- **Criação do aplicativo de mobilidade:**
  - Definir o melhor caminho entre dois pontos da cidade, considerando diferentes meios de transporte.

Com isso definido, podemos começar a planejar a **solução para a licitação**. Bem, aqui não tem muito espaço para inovação, já que precisamos seguir exatamente o que foi pedido. Contudo, no planejamento das linhas de ônibus, temos uma oportunidade de simplificar nossa abordagem.

É de conhecimento geral (ou quase) que a resolução desse tipo de problema se assemelha bastante a problemas NP-completos. Em outras palavras: tentar encontrar a solução ótima aqui é ineficiente e

basicamente uma perda de tempo. Então, em vez disso, vamos focar em uma solução que, mesmo não sendo perfeita, traga a maior satisfação possível para os moradores da cidade.

Para isso, propomos um planejamento que:

- Faça sentido em termos de linhas de ônibus (sem maluquices ou percursos impossíveis).
- Evite muitas voltas desnecessárias.
- Cumpra seu papel de facilitar o deslocamento pela cidade.
- Dê preferência a áreas com atrações e comércios, mas sem a obrigação de passar por todas.

A solução encontrada para esses problemas será descrita e explicada posteriormente.

Agora, vamos olhar para a **criação do aplicativo**. Aqui, temos mais liberdade para tomar decisões criativas que realmente melhorem a experiência do usuário. O ponto de partida é nos colocarmos no lugar de quem usará a nossa solução e entender quais problemas ele realmente precisa resolver.

Pense no seguinte cenário: você quer sair da Rua Getúlio e ir até o Shopping República Velha, que fica do outro lado da cidade. Você ficaria feliz se o caminho sugerido fosse algo assim: pegar um **táxi por 5 km**, andar **a pé por 300 metros**, pegar outro t**áxi por mais 2 km**, depois um **ônibus**, **andar mais um pouco** a pé e, por fim, **outro táxi** para chegar ao destino?

Pois é, eu também não ficaria nada feliz! Especialmente se estivermos falando de uma cidade perigosa ou desconhecida. Então, para evitar esse tipo de situação, nossa ideia é projetar soluções que façam sentido para o usuário e realmente proponham trajetos que ele gostaria de realizar.

Nesse contexto, incluímos, mas não nos limitamos a, decisões como: sempre propor rotas em que a troca de meios de transporte não seja tão frequente e só ocorra quando realmente necessário. Assim, conseguiremos oferecer uma solução que seja útil de verdade, cumpra seu propósito e que faça sentido para o projeto como um todo.

Decisões de desenvolvimento

Voltando um pouco para a realidade, como a cidade Vargas não existe efetivamente, tivemos que criar os gráficos de maneira fictícia. Para isso, resolvemos "emprestar"dados de cidades reais, utilizando como base **Barcelona**, **Berlim**, **Florença** e **Nova York**. Além disso, geradores aleatórios foram usados para definir tanto a quantidade de lotes por segmento quanto a categoria de cada lote. As velocidades das ruas foram extraídas de dados reais, sendo complementadas apenas quando não estavam disponíveis.

Para a API de trânsito, seguimos uma abordagem semelhante: criamos um gerador aleatório que produz um valor percentual indicando a relação entre a velocidade atual do trânsito e a velocidade máxima permitida em cada rua naquele momento. Essa API foi projetada para atualizar todas as arestas do grafo quando chamada, alterando um atributo temporário que representa a situação do trânsito.

Já para os valores financeiros, usamos como referência os preços do Rio de Janeiro. Assim, definimos o valor do metrô em R$7,00 e o ônibus em R$4,50. O táxi, por sua vez, foi configurado com uma tarifa de R$4,00 por quilômetro, com um valor mínimo de R$9,00 por corrida.

Além disso, decidimos que será possível realizar paradas em qualquer esquina ao longo da rota do ônibus. Supomos que essas paradas não causam atrasos significativos no trajeto. Dessa forma, os passageiros poderão embarcar e desembarcar onde preferirem.

2 Projeto das linhas de metrô

Para projetar a linhas de metrô, estruturamos a solução em duas etapas:

1. Primeiro definimos quais vértices serão selecionadas para serem estações. O método que utizamos para definir as estações passa pelas seguintes etapas:
   (a) Utilizamos o algoritmo de Dijkstra para cada região/bairro de forma que tenhamos a distância entre vértices da mesma região;

(b) Após calculado os valores das distâncias de cada vértice pertencentes a mesma região entre si, escolheremos que haja pelo menos uma estação em cada bairro/região. Assim queremos aumentar o máximo possível a proximidade para as demais localidades, então escolheremos o logar mais "central", ou seja, o vértice que tenha a menor distância máxima entre os vértices;

(c) Fazendo isso para cada região teremos selecionado ao menos uma estação por bairro/região, de forma a deixá-la o mais central possível;

2. Com as estações definidas queremos estabelecer a linha no qual o metrô passará. Para definir a linha iremos utilizar uma espécie de algoritmo de Prim Modificado para identificar a Minimal Spanning Tree (MST) que inclua todos esses vértices:

   (a) Inicialmente iremos sobre o gráfo todo estabelecer, através do algoritmo de Dijkstra novamente, o caminho de menor custo que liga cada par de estações;

   (b) Com isso feito, garantimos que haja um caminho de custo mínimo que liga as estações, uma vez que poderia existir um cenário em que as estações escolhidas, apesar de fazerem parte do mesmo grafo (a cidade), não houvesse conexões diretas;

   (c) Com o novo set de vértices para o nosso subgrafo, utilizamos o algoritmo de Prim para achar a MST que irá representar nossa linha de metrô.

Pseudocódigo

Algorithm 1: Adicionar Aresta ao Grafo

Input: Vértices $from$ e to, Bairro $neighborhood$, Comprimento $length$, Velocidade Máxima $maxSpeed$, Direção $oneway$, Número de Lotes $numLotes$, Tipos de Lotes $lotesType$

Adicionar $(from, to, neighborhood, length, maxSpeed, oneway, numLotes, lotesType)$ na lista de adjacências de $from$;
if $oneway = false$ **then**
    Adicionar $(to, from, neighborhood, length, maxSpeed, oneway, numLotes, lotesType)$ na lista de adjacências de to;
end

Algorithm 2: Dijkstra para Encontrar Distâncias

Input: Grafo $G = (V, E)$, vértice inicial $start$
Output: Vetor $dist$ contendo as distâncias mínimas de $start$ para todos os vértices

Inicializar $dist[v] \leftarrow \infty$ para todos $v \in V$, exceto $dist[start] \leftarrow 0$;
Inicializar a fila de prioridade $pq \leftarrow \{(0, start)\}$;
while *pq não está vazia* **do**
    Remover o par $(currentDist, currentNode)$ com menor distância de pq;
    if $currentDist > dist[currentNode]$ **then**
        Continuar para o próximo nó;
    end
    for $edge \in adjList[currentNode]$ **do**
        $newDist \leftarrow currentDist + edge.length$;
        if $newDist < dist[edge.to]$ **then**
            $dist[edge.to] \leftarrow newDist$;
            Adicionar $(newDist, edge.to)$ à pq;
        end
    end
end
return $dist$;

Algorithm 3: Prim para Encontrar a Árvore Geradora Mínima (MST)

Input: Grafo $G = (V, E)$
Output: Árvore Geradora Mínima (MST)

```
Inicializar included ← ∅, pq ← ∅;
Escolher um vértice inicial start ∈ V;
Adicionar todas as arestas incidentes a start na fila de prioridade pq;
included ← {start};
while pq não está vazia do
    Remover o par (peso, (from, to)) com menor peso de pq;
    if to ∉ included then
        Adicionar to a included;
        Adicionar a aresta (from, to) à MST;
        for edge ∈ adjList[to] do
            if edge.to ∉ included then
                Adicionar (edge.length, (to, edge.to)) à pq;
            end
        end
    end
end
return MST;
```

Análise de Complexidade

1. Representação do Grafo

A adição de arestas ao grafo possui complexidade $O(1)$ por inserção em lista de adjacência. Para E arestas, a complexidade total é $O(E)$.

2. Algoritmo de Dijkstra

- O número de vértices é V e o número de arestas é E.

- Cada aresta é relaxada no máximo uma vez. Com o uso de uma fila de prioridade, as operações de inserção e remoção possuem complexidade $O(\log V)$.

- Complexidade total: $O((V + E) \log V)$.

3. Algoritmo de Prim

- A inclusão de cada vértice na árvore custa $O(\log V)$ e cada aresta é processada uma vez.

- Complexidade total: $O((V + E) \log V)$.

Análise de Corretude

- Representação do Grafo: Garante a representação bidirecional para arestas não direcionadas.

- Dijkstra: Utiliza fila de prioridade para garantir a exploração de menores distâncias primeiro, garantindo caminhos mais curtos para todos os vértices.

- Prim: Constrói a MST ao incluir arestas de menor peso que conectam vértices ainda não incluídos.

Discussão dos Resultados

Abaixo estão algumas imagens da execução que mostram a corretude da solução proposta.

```
Esta├║├Áes definidas: 5 6 1
Calculando caminhos mais curtos entre esta├║├Áes...

Encontrando caminho mais curto entre 5 e 6...
  Caminho encontrado: 5 6

Encontrando caminho mais curto entre 5 e 1...
  Caminho encontrado: 5 4 3 2 1

Encontrando caminho mais curto entre 6 e 1...
  Caminho encontrado: 6 0 1
Tempo de execu├║├úo para encontrar todos os caminhos mais curtos: 2ms

Construindo subgrafo conectado...
Tempo de execu├║├úo para construir o subgrafo conectado: 0ms

Gerando MST do subgrafo conectado...
V├®rtice inicial escolhido para a MST: 0
Tempo de execu├║├úo para gerar MST: 0ms
```

```
Validando o grafo:
Representa├║├úo do Grafo:
V├®rtice 6:
  -> 5 (Bairro: Zona Sul, Comprimento: 22m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
V├®rtice 5:
  -> 4 (Bairro: Zona Sul, Comprimento: 8m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
  -> 6 (Bairro: Zona Sul, Comprimento: 22m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
V├®rtice 4:
  -> 3 (Bairro: Zona Sul, Comprimento: 8m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
  -> 5 (Bairro: Zona Sul, Comprimento: 8m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
V├®rtice 3:
  -> 2 (Bairro: Zona Sul, Comprimento: 12m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
  -> 4 (Bairro: Zona Sul, Comprimento: 8m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
V├®rtice 2:
  -> 1 (Bairro: Centro, Comprimento: 15m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
  -> 3 (Bairro: Zona Sul, Comprimento: 12m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
V├®rtice 1:
  -> 0 (Bairro: Centro, Comprimento: 10m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
  -> 2 (Bairro: Centro, Comprimento: 15m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
V├®rtice 0:
  -> 1 (Bairro: Centro, Comprimento: 10m, Velocidade M├íxima: 50km/h, Sentido ├Único: N├úo, N├║mero de Lotes: 1, Tipos de Lotes: [0, 0, 0, 0])
Fim da validacao do grafo.
```

3 Projeto da linha de ônibus

Para projetar a linha de ônibus começamos pensando em como maximizar o número de imóveis comerciais e atrações turísticas no trajeto, e minimizar o número de imóveis residenciais e industriais. Para isso criamos um índice para cada rua da seguinte forma:

$$index = \frac{R + I}{T + C}$$

Onde R é o número de reidências na rua, I é o número de industrias, T o número de atrações turísticas e C o número de inóveis comerciais. Dito isso, as ruas com menor *index* são as melhores para a linha de ônibus passar, pois assim estamos otimizando o número de atrações turísticas e imóveis comerciais atendidos pela linha.

Assim, primeiro selecionamos as ruas com menor *index* de cada bairro, garantindo assim que a linha passará por todos os bairros, e após isso conectamos as arestas de forma que gerem um ciclo. Para conectar as arestas utilizamos o algoritmo Dijkstra, já que queremos encontrar o caminho com menor custo de *index* entre dois pontos.

Pseudocódigos das Funções e Análise da Complexidade

Algorithm 4: SelecionarArestasMinimasPorBairro

Input: Grafo $grafo$
Output: Mapa de arestas mínimas por bairro $arestasMinimasPorBairro$

```
1  Inicializar arestasMinimasPorBairro como um mapa vazio;
2  Inicializar custosMinimosPorBairro como um mapa vazio;
3  for cada vértice u em grafo do
4  |   Obter lista de arestas arestas a partir de u;
5  |   for cada aresta em arestas do
6  |   |   Obter vértice v, bairro bairro, e custo custo;
7  |   |   if bairro não existe em arestasMinimasPorBairro ou
   |   |    custo < custosMinimosPorBairro[bairro] then
8  |   |   |   Atualizar arestasMinimasPorBairro[bairro] ← (u, v);
9  |   |   |   Atualizar custosMinimosPorBairro[bairro] ← custo;
10 |   |   end
11 |   end
12 end
13 return arestasMinimasPorBairro;
```

No pseudocódigo acima podemos ver que iteramos para todos os vértices e depois para todas as arestas, logo temos $O(V + E)$.

Algorithm 5: ReconstruirCaminho

Input: Vértice *origem*, vértice *destino*, vetor *parent*
Output: Vetor *caminho* com a sequência de vértices

```
1  Inicializar caminho como vetor vazio;
2  v começando em destino;
3  while v ≠ origem do
4  |   Adicionar v ao início de caminho;
5  |   Atualiza v para parent[v];
6  end
7  Adicionar origem ao início de caminho;
8  return caminho;
```

Já neste segundo algoritmo iteramos por todos os vértices entre o inicio e o fim, o que no pior caso vai ser $O(V)$.

Algorithm 6: EncontrarCicloArestasMinimas

Input: Grafo $grafo$, mapa $arestasMinimas$
Output: Vetor *ciclo* representando o ciclo fechado

```
1  Inicializar ciclo como vetor vazio;
2  Inicializar arestasAdicionadas como conjunto vazio;
3  for cada vértice em arestasMinimas do
4  |   Adicionar vértices u e v ao ciclo, se ainda não presentes;
5  |   Adicionar aresta (u, v) ao conjunto arestasAdicionadas;
6  end
7  for i de 0 até |ciclo| − 2 do
8  |   Obter vértices u e v consecutivos no ciclo;
9  |   if (u, v) ∉ arestasAdicionadas then
10 |   |   Executar Dijkstra para encontrar menor caminho entre u e v;
11 |   |   Reconstruir caminho caminho entre u e v;
12 |   |   Inserir vértices de caminho no ciclo;
13 |   end
14 end
15 Conectar último vértice do ciclo ao primeiro;
16 return ciclo;
```

Por último, temos o algoritmo que gera a rota do ônibus. Começando pelo primeiro for iteramos sobre todos os vértices das arestas mínimas, como temos uma aresta por bairro, aqui iteramos sobre $2b$ vértices, onde b é o número de bairros do grafo.

No segundo for iteramos sore o número de vértices no ciclo($O(V)$ no pior caso), executamos um Dijkstra para cada um deles ($O((V+E)logV)$), depois construimos o caminho ($O(V)$) e inserimos os vértices do caminho no ciclo ($O(V)$). Com isso, Para esse bloco temos complexidade $O(V(V+E)logV)$ no pior caso. Depois so precisamos inserir o primeiro vértice no ciclo.

Logo esse algoritmo tem complexidade $O(V(V+E)logV)$.

Análise da corretude do algoritmo

Para isso é preciso analisar se o algoritmo atende as solicitações feitas pela prefeitura da cidade de Vargas.

- Passar por todos os bairros: Garantimos que a linha de ônibus irá passar por todos os bairros pois começamos selecionando uma aresta de cada bairro e depois conectamos elas.
- Maximizar o número de imóveis comerciais e atrações turísticas no trajeto e minimizar o número de imóveis residenciais e industriais: Para que isso ocorra implementamos um índice para cada rua relecionando os 4 tipos de imóveis possiveis e no processo de conectar as arestas selecionadas utilizamos um algoritmo de menor caminho, assim garantimos que esse item também seja cumprido.

Discussão dos resultados do experimento

Dentro dos tamanhos que testamos para diferentes gráficos percebemos um aumento no tempo de execução conforme a evolução do mesmo. O que se alinha com o esperado e calculado da complexidade do código.

Versão 2 do algoritmo

Para aprimorar ainda mais nossa abordagem e nos aproximarmos da solução ótima, decidimos modelar o problema de forma a possibilitar sua resolução utilizando força bruta. Partimos da mesma lógica do algoritmo anterior para selecionar as arestas com os menores coeficientes e, a partir delas, identificamos os vértices iniciais. Com esses vértices, criamos um subgrafo totalmente conectado, onde cada vértice representa um bairro.

Para estabelecer as conexões no subgrafo, utilizamos o algoritmo de Dijkstra para encontrar os melhores caminhos entre os vértices e assim definir as novas arestas do subgrafo. O resultado é um grafo significativamente reduzido, contendo apenas o número de bairros como vértices.

Por fim, aplicamos força bruta para resolver o problema do caixeiro viajante nesse subgrafo reduzido, devido ao pequeno número de vértices, o que torna essa abordagem computacionalmente viável.

Pseudocódigos das Funções e Análise da Complexidade

Aqui, reutilizamos muita coisa, então irei apenas citar.

Algorithm 7: AcharCaminhoOnibus

Input: Grafo $grafo$, vetor $verticesMinimos$
Output: Vetor $ciclo$ representando o ciclo fechado

Inicializar $subgrafo$; **for** *cada* $vértice_i$ *em* $verticesMinimos$ **do**
    Executar Dijkstra para encontrar a SPT;
    for *cada* $vértice_j$ *em* $verticesMinimos$ **do**
        Se i == j, pule para o proximo; Ache a *distancia* entre $vértice_i$ e $vértice_j$; Adicione ($vértice_i$, $vértice_j$, *distancia*) ao subgrafo;
    end
end
Use força bruta para resolver o caixeiro viajante no $subgrafo$, resultando no sub_rota;
Inicializar $caminho_final$; **for** *cada vértice em* sub_rota **do**
    Use dijkstra para reconstruir o caminho de *vértice* até $vértice.next()$ no grafo completo ($caminho_parcial$); Adicione $caminho_parcial$ ao final de $caminho_final$
end
return $caminho_final$;

Aqui, lidamos com complexidades tanto polinomiais quanto não polinomiais. Inicialmente, aplicamos o algoritmo de Dijkstra para cada bairro, resultando em uma complexidade de $O(|B| \cdot v \log(v))$. Na segunda parte polinomial, aplicamos novamente o algoritmo de Dijkstra para cada bairro, com a mesma complexidade $O(|B| \cdot v \log(v))$.

Entretanto, o maior desafio está na resolução do problema do caixeiro viajante por força bruta, que possui complexidade $O(n!)$. No entanto, no nosso caso, limitamos n ao número de bairros, reduzindo a complexidade para $O(|B|!)$, o que torna a abordagem mais viável.

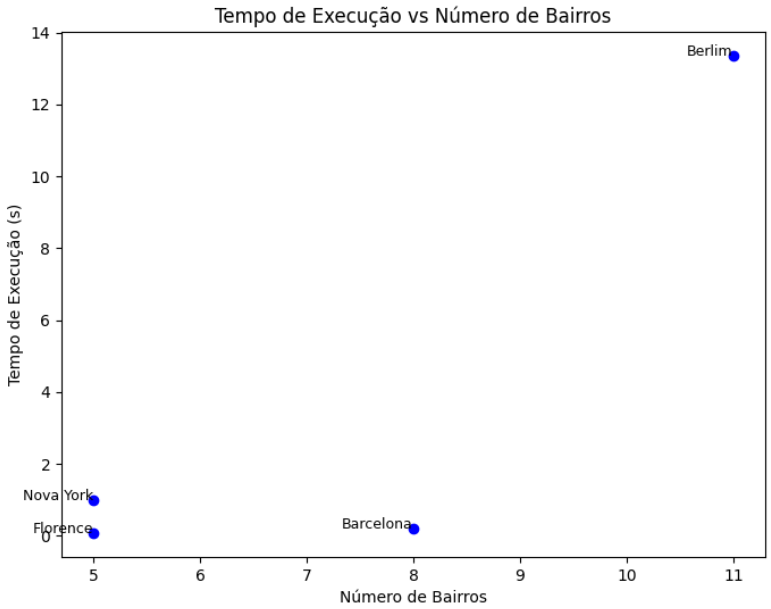


Figura 1: Aqui, fica evidente o fatorial!

4 Criação de rotas

Definidas as linhas de metrô e ônibus, vamos partir para o problema de achar uma rota eficiente entre dois endereços, dado um gasto máximo. Primeiro vamos começar com algumas considerações para embasar nossa proposta de busca de uma rota eficiente.

Uma pessoa só consegue andar de ônibus ou metro se já estiver em uma estação. O único jeito de andar livremente nas ruas é de taxi, que considera o sentido das ruas, e a pé, que anda em qualquer sentido. Para

chegar em uma estação (metrô ou ônibus), se a pessoa tiver dinheiro suficiente, pensando em minimizar o tempo sempre será preferível usar um taxi. Portanto, ao andarmos livremente podemos considerar que uma boa escolha é usar o máximo de dinheiro possível no taxi e andar o restante.

O metrô é considerado uma forma eficiente de locomoção. Sua alta velocidade constante faz com que, se linha estiver nos aproximando do destino final, valha a pene pegar ele. Entretano, podemos considerar que dificilmente, em uma rota eficiente, pegaremos o metrô duas vezes.

Como uma forma de simplificação, podemos considerar que existem algumas combinações possíveis de caminho que podem ser eficientes para chegar ao destino. Buscando a melhor rota de cada uma das seguintes combinações, podemos encontrar uma maneira eficiente de chegar ao destino:

- Andar livremente: Andar o máximo possível de taxi em direção ao destino final e terminar a pé.
- Metrô: andamos livremente até a estação de metrô mais próxima, descemos na estação mais próxima do destino e andamos livremente até ele. Guardamos o dinheiro suficiente para uma passagem de metrô e gastamos o resto em taxi nos trajetos de andar livremente.
- Ônibus: andamos livremente até a estação de ônibus mais próxima, descemos na estação mais próxima do destino e andamos livremente até ele. Guardamos o dinheiro suficiente para uma passagem de metrô e gastamos o resto em taxi nos trajetos de andar livremente.
- Metrô + ônibus: nesse caso, consideramos pegar o ônibus para chegar a estação de metro mais próxima e também para ir da estação de descida para o destino final.

Para cada uma das combinações listadas acima, vamos implementar algoritmos para achar a melhor, ou uma boa maneira, de encontrar uma rota. Achando o melhor de cada uma das 4 combinações, escolhemos o melhor para retornar ao usuário.

Andar livremente

Dado um gasto máximo, queremos gastar todo o nosso dinheiro com taxi e, se ainda for necessário, andar o restante do caminho. Para isso criamos as seguintes funções:

Algorithm 8: Calcula Tempo Taxi

Input: Grafo $grafo$, vertex $origem$, vertex $destino$, vetor $parentTaxi$
Output: float $tempo$

```
1  for cada aresta em graph.getEdges(parentTaxi[vertice]) do
2      if aresta conecta parentTaxi[vertice] a vertice then
3          Calcular cost, v_max e modf_transito;
4          Atualizar tempo ← tempo + cost/(v_max · modf_transito);
5          encontrou ← verdadeiro;
6          break;
7      end
8  end
9  if not encontrou then
10     Exibir mensagem de erro e break;
11 end
12 Atualizar vertice ← parentTaxi[vertice];
```

Na função estamos percorrendo o caminho utilizado pelo taxi de trás para frente e olhando todas as arestas de cada vértice do caminho, por isso, o algoritmo apresenta complexidade de $\Theta(V + E)$ no pior caso, que para um grafo esparso fica $\Theta(V)$. Porém, na realidade será menor, visto que não é comum um caminho ter um número de vértices proporcional ao número total.

Algorithm 9: Calcula Tempo Caminhada

Input: Grafo $grafo$, vertex $origem$, vertex $destino$, vetor $parentCaminhada$
Output: float $tempo$

```
for cada aresta em graph.getEdges(parentCaminhada[vertice]) do
    if aresta conecta parentCaminhada[vertice] a vertice then
        Calcular cost;
        Atualizar tempo ← tempo + cost/5;
        encontrou ← verdadeiro;
        break;
    end
end
if not encontrou then
    Exibir mensagem de erro e break;
end
Atualizar vertice ← parentCaminhada[vertice];
```

Dadas as mesmas considerações do algoritmo anterior, ele também apresenta complexidade de $\Theta(V)$ no pior caso.

Algorithm 10: Acha rota livre

```
Inicializar vertice ← destino, dist ← 0;
Inicializar caminhoTaxi como lista vazia;
while vertice ≠ parentTaxi[vertice] do
    if dist < distMax e parada ainda não foi definida then
        Adicionar vertice ao caminhoTaxi;
        Atualizar parada ← vertice;
    end
    Atualizar vertice ← parentTaxi[vertice];
end
Adicionar origem ao caminhoTaxi;
if parada == destino then
    Calcular tempoTaxi ← CalculaTempoTaxi(graphDirecionado, origem, destino, parentTaxi);
    Adicionar caminhoTaxi à rota;
    return tempoTaxi;
end
tempoTaxi ← CalculaTempoTaxi(graphDirecionado, origem, parada, parentTaxi);
Adicionar caminhoTaxi à rota;
```

A complexidade da função rotaTaxi depende principalmente da execução do algoritmo de Dijkstra, que tem complexidade $O((V+E)logV)$. O Dijkstra é executado duas vezes: uma no grafo direcionado (para calcular o caminho de táxi) e outra no grafo completo (para o caminho a pé). Além disso, há um loop para reconstrução dos caminhos, que percorre no máximo $O(V)$. Assim, a complexidade geral da função é $O((V+E)logV)$, dominada pelas execuções do Dijkstra.

Ônibus

A ideia do trajeto que utiliza ônibus é andar livremente até o ponto de ônibus mais perto e descer no ponto mais perto do destino. Desse ponto andamos livremente até o destino. Para isso vamos reaproveitar as funções feitas acima para andar livremente.

Algorithm 11: RotaOnibus

Input: *graphCompleto*, *graphDirecionado*, vetor *linhaOnibus*, lista *rota*, inteiro *origem*, *destino*, real *custoMax*

Output: Tempo total da rota utilizando ônibus e táxi

Definir constante $CUSTO\_ONIBUS \leftarrow 4.50$;
if $custoMax < CUSTO\_ONIBUS$ **then**
  return $+\infty$;
end
$custoMax \leftarrow custoMax - CUSTO\_ONIBUS$;
Executar Dijkstra em *graphCompleto* a partir de *origem*;
Salvar *distanceOrigem* e *parentOrigem*;
Executar Dijkstra em *graphCompleto* a partir de *destino*;
Salvar *distanceDestino* e *parentDestino*;
Inicializar $embarque \leftarrow -1$, $desembarque \leftarrow -1$;
Inicializar $menorDistOrigem \leftarrow +\infty$, $menorDistDestino \leftarrow +\infty$;
for *cada vértice v em linhaOnibus* **do**
  if $distanceOrigem[v] < menorDistOrigem$ **then**
    $menorDistOrigem \leftarrow distanceOrigem[v]$;
    $embarque \leftarrow v$;
    $idx\_embarque \leftarrow$ índice(v);
  end
  if $distanceDestino[v] < menorDistDestino$ **then**
    $menorDistDestino \leftarrow distanceDestino[v]$;
    $desembarque \leftarrow v$;
    $idx\_desembarque \leftarrow$ índice(v);
  end
end
if $embarque = -1$ *ou* $desembarque = -1$ **then**
  return $+\infty$;
end
Inicializar *caminhoOnibus* como lista vazia;
$i \leftarrow idx\_embarque$;
while $i \neq idx\_desembarque$ **do**
  Adicionar $linhaOnibus[i]$ a *caminhoOnibus*;
  $i \leftarrow (i + 1) \bmod |linhaOnibus|$;
end
Adicionar $linhaOnibus[i]$ a *caminhoOnibus*;
Inicializar $tempoIda \leftarrow 0$, $tempoOnibus \leftarrow 0$, $tempoSaida \leftarrow 0$;
if $embarque \neq desembarque$ **then**
  $tempoIda \leftarrow rotaTaxi(graphCompleto, graphDirecionado, rota, origem, embarque, custoMax)$;
  $tempoOnibus \leftarrow calculaTempoOnibus(graphDirecionado, caminhoOnibus)$;
  Adicionar BUS_CODE, *tempoOnibus* e *caminhoOnibus* a *rota*;
  $tempoSaida \leftarrow$
   $rotaTaxi(graphCompleto, graphDirecionado, rota, desembarque, destino, custoMax)$;
end
else
  $tempoSaida \leftarrow rotaTaxi(graphCompleto, graphDirecionado, rota, origem, destino, custoMax)$;
end
return $tempoIda + tempoOnibus + tempoSaida$;

A proposta do algoritmo é utilizar dijkstra a partir do vértice de origem para encontrar a estação de ônibus mais próxima. Dessa estação andamos de ônibus até a estação mais próxima do destino, também encontrada com dijkstra. Daí, vamos ao destino final. Para andar sem ser de ônibus utilizamos a funçaõ criada anteriormente, rotaTaxi, que anda de taxi ou a pé de acordo com o dinheiro.

A complexidade da função `rotaOnibus` é dominada pelas execuções do algoritmo de Dijkstra, que possuem complexidade $O(V + E \log V)$, onde V é o número de vértices e E é o número de arestas do grafo. Esses cálculos são realizados duas vezes, uma para a origem e outra para o destino. Além disso, há uma

iteração sobre os vértices da linha de ônibus ($O(L)$), onde L é o tamanho da linha de ônibus, para encontrar os vértices de embarque e desembarque mais próximos. A reconstrução do caminho do ônibus percorre no máximo L vértices, resultando em $O(L)$. Assim, a complexidade total é aproximadamente $O(2(V + E \log V) + L)$, sendo a parte $V + E \log V$ a mais significativa devido ao uso de Dijkstra.

Melhor Rota

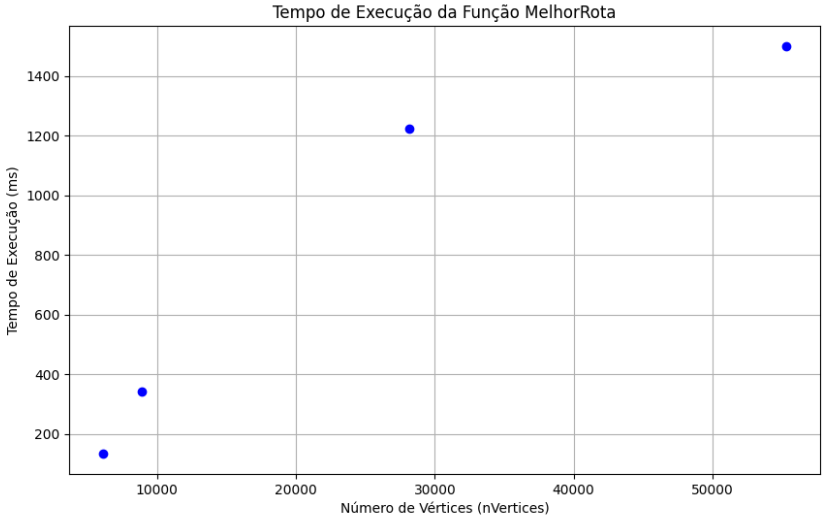
Como já foi dito, para acharmos a melhor rota vamos achar a menor entre os tipos de rota definidos acima.

Algorithm 12: Encontrar a Melhor Rota

Inicializar $rota_1$ e $rota_2$ como listas vazias;
Inicializar $resto \leftarrow 0$, $horaAtual \leftarrow getHoraAtual()$;
Atualizar o trânsito no grafo usando $TrafficAPI$;
Criar $graphCompleto$ como cópia bidirecional de $graph$;
Criar $graphDirecionado$ como cópia de $graph$;
Selecionar as arestas mínimas por bairro em $graph$;
Encontrar $ciclo \leftarrow encontrarCicloArestasMinimas(graph, arestasMinimas)$;
Calcular $tempoTaxi \leftarrow$
 $rotaTaxi(graphCompleto, graphDirecionado, rota_1, origem, destino, custoMax, resto)$;
Calcular $tempoOnibus \leftarrow$
 $rotaOnibus(graphCompleto, graphDirecionado, ciclo, rota_2, origem, destino, custoMax, horaAtual)$;

if $tempoTaxi < tempoOnibus$ **then**
 Imprimir $rota_1$;
end
else
 Imprimir $rota_2$;
end
return;

A complexidade esperada para retornar a melhor rota para o usuário é de $\Theta(V \log V)$ no pior caso, que é a complexidade dos algoritmos descritos acima e essa função, basicamente, roda eles e utiliza o que teve melhor performance. Mas vamos observar o na prática como ele se comporta.



Como podemos observar, o gráfico apresenta um comportamento de uma função $\log V$, apesar da complexidade esperada ser maior. Isso se dá, provavelmente, porque o cálculo para complexidade esperada leva em conta o pior caso.