Projeto e Análise de Algoritmos

Kauan Mariani Ferreira
Lívia Machado S. Verly
Matheus Fillype Ferreira de Carvalho
Pedro Henrique Coterli
Sillas Rocha da Costa

Relatório apresentado para a matéria Projeto e Análise de Algoritmos misnistrada pelo Doutor Thiago Pinheiro de Araújo

Escola de Matemática Aplicada
Fundação Getúlio Vargas
Rio de Janeiro, RJ - Brasil
5 de dezembro de 2024

Sumário

1 Introdução

Neste relatório, apresentamos uma abordagem para a resolução de problemas arquiteturais e de roteamento utilizando representações gráficas baseadas em dados reais da cidade de Barcelona, na Espanha. O trabalho foi estruturado em três principais tarefas: o projeto de linhas de metrô eficientes, o planejamento de uma rota de ônibus hop-on/hop-off que maximize a presença de pontos de interesse e o cálculo de rotas mais rápidas.
Para tanto, desenvolvemos duas bases de código, `estrutura.cpp` e `algoritmosBase.cpp`, que fornecem as ferramentas necessárias para modelar e resolver os problemas propostos. Estas bases incluem estruturas de dados otimizadas, como listas de adjacências, e implementações de algoritmos clássicos, como Dijkstra e Prim, garantindo tanto a corretude quanto a eficiência das soluções desenvolvidas. Os resultados obtidos demonstram a viabilidade das técnicas aplicadas e a capacidade dos algoritmos de atender às especificações do problema.
A implementação do trabalho se encontra em: https://github.com/kauanmaf/trabalho_paa

2 Modelagem Arquitetural Geral

Para a realização das tarefas, decidimos criar um mapa e representá-lo por um grafo utilizando os dados de Barcelona, cidade da Espanha. Foram criados documentos base que foram utilizados em todas as tarefas: `estrutura.cpp` e `algoritmosBase.cpp`.

2.1 Estrutura de Dados (`estrutura.cpp`)

Este arquivo define as principais classes e estruturas necessárias para modelar os problemas, incluindo:

- **Classe Planta:** Representa um grafo usando uma lista de adjacência. Permite a inserção de vértices e arestas, bem como a manipulação de informações associadas ao grafo.
- **Classe Segmento:** Modela as arestas do grafo, incluindo atributos como limite de velocidade, comprimento e informações regionais (e.g., CEP, nome da rua).
- **Classe Imóvel:** Representa os nós ou informações associadas a cada ponto do grafo, como tipo de imóvel e localização relativa.

2.2 Algoritmos (`algoritmosBase.cpp`)

Este arquivo implementa os algoritmos principais para resolução dos problemas propostos:

- **Dijkstra:** Calcula o menor caminho a partir de um vértice inicial para todos os outros no grafo. Utiliza uma fila de prioridade para garantir eficiência.
- **Prim:** Determina a árvore geradora mínima de um grafo (MST). Utiliza também uma fila de prioridade para selecionar arestas com menor custo.

3 Tarefa 1

A Tarefa 1 tem como objetivo projetar as linhas de metrô da cidade considerando que todas as regiões possuam estações localizadas estrategicamente em cruzamentos, de forma a minimizar a distância para o ponto mais distante de cada região. O projeto define os segmentos de escavação necessários para conectar todas as estações, garantindo que o custo total para a cidade seja o menor possível, sem a necessidade de especificar rotas de trens.

3.1 Modelagem arquitetural

O problema foi modelado como um grafo, em que os vértices representam cruzamentos da cidade e as arestas representam os segmentos de rua, cada qual com um custo associado para escavação. As regiões da cidade foram representadas por conjuntos de vértices associados a um mesmo CEP. O objetivo principal da modelagem é selecionar os segmentos de menor custo que conectem todas as estações, garantindo que cada região tenha pelo menos uma estação, enquanto se minimiza a distância máxima entre o ponto mais distante e a estação daquela região.

3.2 Estrutura de dados

A estrutura de dados central é um grafo, representado por uma lista de adjacências onde cada vértice aponta para suas arestas correspondentes. A classe `Planta` encapsula o grafo, armazenando a lista de adjacências e o conjunto de CEPs que definem as regiões.

Além disso, foi utilizada uma matriz booleana para rastrear quais vértices pertencem a quais regiões. Vetores auxiliares foram empregados para suportar o cálculo de distâncias e a seleção de vértices estratégicos:

- `minMaxDistances`: armazena as menores distâncias máximas para cada região.
- `minMaxDistancesVertices`: identifica o vértice que corresponde à menor distância máxima em cada região.
- `parents` e `distances`: usados no algoritmo de Dijkstra para rastrear os caminhos e calcular as distâncias mínimas de cada vértice.

3.3 Pseudo-código

Tarefa 1:

```
plantaND = Planta()

for v de 0 a |V| - 1:
    edges = planta -> listaAdj[v]
    for aresta edge em edges:
        Adiciona edge a plantaND -> listaAdj[v]
        if edge -> dupla == True:
            continue
        else:
```

```
            edge2 = newSegmento(vSaida = edge -> vEntrada,
                                vEntrada = edge -> vSaida,
                                limVel = edge -> limVel,
                                tamanho = edge -> tamanho,
                                CEP = edge -> CEP,
                                rua = edge -> rua,
                                dupla = True)
            adicionaSegmentoAPlanta(edge2, plantaND)

regioes = vetor de tamanho |planta -> CEPs|
for i de 0 a |planta -> CEPs| - 1:
    regiao = vetor de tamanho |V|
    for j de 0 a |V| - 1:
        regiao[j] = False
    regioes[i] = regiao

for v de 0 a |V| - 1:
    edges = plantaND -> listaAdj[v]
    for aresta edge em edges:
        regioes[edge -> CEP][v] = True


min_max_distances, min_max_distances_vertices,
↪  min_max_distances_parents, min_max_distances_length = vetores de
↪  tamanho |planta -> CEPs|
for i de 0 a |planta -> CEPs| - 1:
    min_max_distances[i] = infinito
    min_max_distances_vertices[i] = -1
    temp_parents = vetor de tamanho v
    for j de 0 a tamanho |V|-1:
        temp_parents[j] = -1
    min_max_distances_parents[i] = temp_parents

    temp_distances = vetor de tamanho v
    for j de 0 a tamanho |V|-1:
        temp_distances[j] = INT_MAX
    min_max_distances_length[i] = temp_distances


for v de 0 a |V| - 1:
    parents, distances = vetores de tamanho |V|
    for i de 0 a |V| - 1:
        parents[i] = -1
        distances[i] = infinito

    Dijkstra(v, parents, distances, |V|, plantaND)

    for regiao em regioes:
        if regiao[v] == True:
```

```
            max_distance = 0
            max_distance_vertice = -1
            for vértice v2 de 0 a |V| - 1:
                if regiao[v2] == True and distances[v2] > max_distance:
                    max_distance = distances[v2]
                    max_distance_vertice = v2
            if max_distance < min_max_distances[regiao]:
                min_max_distances[regiao] = max_distance
                min_max_distances_vertices[regiao] = v
                min_max_distances_parents[regiao] = parents
                min_max_distances_length[regiao] = distances

plantaVirtual = Planta()

for i de 0 a |planta -> CEPs|-1:
    for j de 0 a |planta -> CEPs|:
        if i != j:
            v_saida = min_max_distances_vertices[i],
            v_entrada = min_max_distances_vertices[j]
            newSegmento(vSaida = i,
                        vEntrada = j,
                        tamanho = min_max_distances_length[i][v_entrada])

for v de 0 a |planta -> CEPs| - 1:
    parents = vetores de tamanho |V|
    for i de 0 a |V| - 1:
        parents[i] = -1

MST(parents, 0, plantaVirtual)

result = lista de arestas vazia

for i de 1 a |planta -> CEPs| - 1:
    virtual_parent = parents[i]
    real_parent = min_max_distances_vertices[virtual_parent]
    current_real_parents = min_max_distances_parents[virtual_parent]
    real_start = min_max_distances_vertices[i]

    while real_start != real_parent:
        Adiciona (current_real_parents[real_start], real_start) ao result
        real_start = current_real_parents[real_start]

return result

```

Algorithm 1 Parte 1 - Preparação de Dados e Inicialização

Data: Planta com lista de adjacências e CEPs
Result: Planta com segmentos e lista de regiões
Função `main`**:**

```
plantaND = Planta()  for v = 0 to |V| − 1 do
    edges = planta.listaAdj[v]  foreach edge em edges do
        Adiciona edge a plantaND.listaAdj[v]  if edge.dupla == True then
            continue
        end
        else
            edge2 = newSegmento(vSaida = edge.vEntrada, vEntrada =
             edge.vSaida, limVel = edge.limVel, tamanho = edge.tamanho,
             CEP = edge.CEP, rua = edge.rua, dupla = True)  adicionaSeg-
             mentoAPlanta(edge2, plantaND)
        end
    end
end
regioes = vetor de tamanho |planta → CEPs|  for i = 0 to |planta → CEPs|−1
 do
    regiao = vetor de tamanho |V|  for j = 0 to |V| − 1 do
        regiao[j] = False
    end
    regioes[i] = regiao
end
```

3.4 Corretude

A seguir, validamos a corretude considerando a especificação do problema, a prova de fim e a correção parcial:

Prova de fim: O algoritmo sempre termina porque realiza operações finitas em estruturas de dados bem definidas.

- O cálculo das menores distâncias em cada região usa o algoritmo de Dijkstra, que processa cada vértice uma vez.
- A construção do grafo virtual e a aplicação do algoritmo de Prim para encontrar a árvore geradora mínima ocorrem em um número finito de passos.

Correção parcial: Se o algoritmo termina, ele produz a solução correta devido aos seguintes fatores:

- Dentro de cada região, o vértice escolhido é aquele que minimiza a distância máxima para todos os outros vértices, de acordo com o resultado do algoritmo de Dijkstra. Isso garante a melhor localização possível da estação.
- A conexão entre as regiões utiliza o grafo virtual, onde cada aresta representa o menor custo entre duas regiões. A aplicação do algoritmo de Prim assegura que o custo total de escavação seja minimizado.

Algorithm 2 Parte 2 - Cálculo das Distâncias Máximas

Data: Planta com lista de adjacências e CEPs
Result: Distâncias máximas para cada região
Função `main`**:**

```
min_max_distances, min_max_distances_vertices,
 min_max_distances_parents, min_max_distances_length = vetores de
 tamanho |planta → CEPs|  for i = 0 to |planta → CEPs| − 1 do
    min_max_distances[i] = infinito  min_max_distances_vertices[i] = -1
     temp_parents = vetor de tamanho |V|  for j = 0 to |V| − 1 do
        temp_parents[j] = -1
    end
    min_max_distances_parents[i] = temp_parents
    temp_distances = vetor de tamanho |V|  for j = 0 to |V| − 1 do
        temp_distances[j] = INT_MAX
    end
    min_max_distances_length[i] = temp_distances
end
for v = 0 to |V| − 1 do
    parents, distances = vetores de tamanho |V|  for i = 0 to |V| − 1 do
        parents[i] = -1  distances[i] = infinito
    end
    Dijkstra(v, parents, distances, |V|, plantaND)  foreach região em regioes
     do
        if região[v] == True then
            max_distance = 0  max_distance_vertice = -1  for v2 = 0 to |V|−1
             do
                if região[v2] == True and distances[v2] ¿ max_distance then
                    max_distance = distances[v2]  max_distance_vertice = v2
                end
            end
            if max_distance ¡ min_max_distances[região] then
                min_max_distances[região] = max_distance
                 min_max_distances_vertices[região] = v
                 min_max_distances_parents[região] = parents
                 min_max_distances_length[região] = distances
            end
        end
    end
end
```

Algorithm 3 Parte 3 - Construção do MST e Resultado Final

Data: Planta com segmentos e distâncias
Result: MST e resultado final com arestas
Função `main`:

```
plantaVirtual = Planta()  for i = 0 to |planta → CEPs| − 1 do
    for j = 0 to |planta → CEPs| − 1 do
        if i ≠ j then
            v_saida = min_max_distances_vertices[i]  v_entrada = min_max_distances_vertices[j]  newSegmento(vSaida = i, vEntrada = j, tamanho = min_max_distances_length[i][v_entrada])
        end
    end
end
for v = 0 to |planta → CEPs| − 1 do
    parents = vetores de tamanho |V|  for i = 0 to |V| − 1 do
        parents[i] = -1
    end
    MST(parents, 0, plantaVirtual)
end
result = lista de arestas vazia  for i = 1 to |planta → CEPs| − 1 do
    virtual_parent = parents[i]  virtual_parent_edge = plantaVirtual.listaAdj[virtual_parent][i]  Adiciona virtual_parent_edge a result
end
return result
```

A corretude do algoritmo é assegurada pela combinação da terminação garantida e da lógica correta de escolha de estações e conexões, atendendo às especificações do problema de forma eficiente.

3.5 Complexidade

- **Função:** `plantaND = Planta()`
  Cria uma cópia não direcionada do grafo original, adicionando as arestas de forma bidirecional. A complexidade dessa operação é $O(V + E)$, pois percorre os vértices e as arestas da planta original.

- **Função:** `for v de 0 a |V| - 1`
  Para cada vértice, verifica as arestas e adiciona as arestas na direção oposta, caso não sejam duplas. Isso tem uma complexidade $O(V + E)$, onde V é o número de vértices e E é o número de arestas.

- **Função:** `for i de 0 a |planta -> CEPs| - 1`
  Cria uma matriz que indica quais vértices possuem arestas que pertencem a uma região específica (CEP). A complexidade é $O(E \cdot V + V + E) = O(VE)$.

- **Função:** `for v de 0 a |V| - 1`
  Para cada vértice, percorre as arestas e atualiza a matriz de regiões, indicando a pertença de cada vértice a uma região. A complexidade é $O(VE)$.

- **Função:** `Dijkstra(v, parents, distances, |V|, plantaND)`
  Executa o algoritmo de Dijkstra para cada vértice, calculando as distâncias até os outros vértices, a partir do vértice de origem. A complexidade é $O(V^2+(V+E)\cdot\log V+E\cdot V)=O(V^3)$.

- **Função:** `for v de 0 a |V| - 1`
  Para cada vértice, executa o algoritmo de Dijkstra e, com base nas distâncias obtidas, determina o vértice mais distante em cada região. A complexidade é $O(V^3)$, pois envolve rodar o algoritmo de Dijkstra para cada vértice e calcular as distâncias para todos os vértices das regiões.

- **Funções de distância**
  Calcula as distâncias mínimas e as regiões mais distantes para cada vértice. A complexidade total dessa operação é $O(V^3)$, já que envolve rodar o algoritmo de Dijkstra para cada vértice.

- **Função:** `plantaVirtual = Planta()`
  Cria uma planta virtual completa, incluindo arestas entre os vértices que terão estações, com os tamanhos dos caminhos mais curtos. A complexidade é $O(E\cdot E+E\cdot V+(V+E)\cdot\log V+E\cdot V)=O(V^2)$.

- **Função:** `MST(parents, 0, plantaVirtual)`
  Encontra a Árvore Geradora Mínima (MST) da planta virtual usando o algoritmo apropriado. A complexidade é $O(V^2)$.

- **Função:** `for i de 1 a |planta -> CEPs| - 1`
  Para cada vértice, traça o caminho mais curto até os outros vértices utilizando a árvore MST e as distâncias mínimas calculadas. A complexidade é $O(V^2)$.

- Complexidade Total A complexidade total do algoritmo pode ser determinada pela soma das complexidades das funções principais. As funções com maior complexidade dominam o tempo de execução.

  Portanto, a complexidade total do algoritmo é:

  $$O(V^3)$$

3.6 Resultados

3.6.1 Desempenho Computacional

Para avaliarmos o desempenho computacional do nosso algoritmo, realizamos testes com diferentes configurações de entrada, variando o número de vértices (V) e arestas (E) de grafos. As entradas foram definidas em três categorias de densidade, baseadas na relação entre o número de arestas e vértices, assumindo os valores 1, 2 e 3. Os tempos de execução obtidos foram registrados em nanosegundos, conforme apresentado na Figura 1.

Os resultados mostram que o tempo de execução do algoritmo aumenta conforme o tamanho da entrada cresce, principalmente com o aumento do número de vértices. Observou-se que:

- O aumento do número de vértices sempre acarreta em um maior tempo de execução.

Figura 1: Tempos de execução em diferentes configurações de entrada.

	problem	V	E	ratio	time
0	1	100	100	1	58717000
1	1	100	200	2	45054000
2	1	100	294	3	26777000
3	1	300	300	1	258480000
4	1	300	600	2	214917000
5	1	300	894	3	657358000
6	1	500	500	1	855895000
7	1	500	1000	2	736911000
8	1	500	1494	3	835982000
9	1	700	700	1	1700262000
10	1	700	1400	2	2639048000
11	1	700	2094	3	1428587000
12	1	900	900	1	3203909000
13	1	900	1800	2	2329632000
14	1	900	2694	3	2943897000

- O aumento no número de arestas nem sempre provoca um aumento no tempo de execução. Pelo contrário, observou-se que, quanto maior o número de vértices, mais arestas são necessárias para reduzir o tempo de execução. No geral, os casos classificados com densidade 2 e 3 apresentaram tempos de execução satisfatórios.

Para complementar as análises, plotamos um gráfico de linhas que avalia a relação entre o tempo de execução e o número de vértices.

O comportamento do tempo de execução indica um crescimento que pode ser aproximadamente quadrático ou cúbico em relação ao tamanho do grafo. Essa tendência é mais pronunciada nos casos em que o tempo quase triplica conforme o número de vértices aumenta. É importante ressaltar que o tempo de execução de grafos mais densos tende a ser menor à medida que o número de vértices aumenta. Isso pode ser observado no comportamento das linhas azul (menos denso) e verde (mais denso) no gráfico.

Os testes realizados confirmaram que o algoritmo é funcional para diversas configurações de entrada, sendo ainda mais eficiente em grafos densos.

3.6.2 Exemplo Prático

Também realizamos a implementação de um exemplo prático para validar o funcionamento do algoritmo.

Na Figura 3, observamos que o grafo está dividido em cores, onde cada cor representa uma região. A linha preta representa uma linha de metrô criada pelo algoritmo, e cada vértice preto indica uma estação, localizada em cada região.

Com isso, é possível verificar que o algoritmo funciona na prática, pois conseguiu criar uma linha de metrô que minimiza a distância para o ponto mais distante de cada região, atendendo ao objetivo estabelecido.

Figura 2: Relação entre o tempo de execução e o número de vértices.

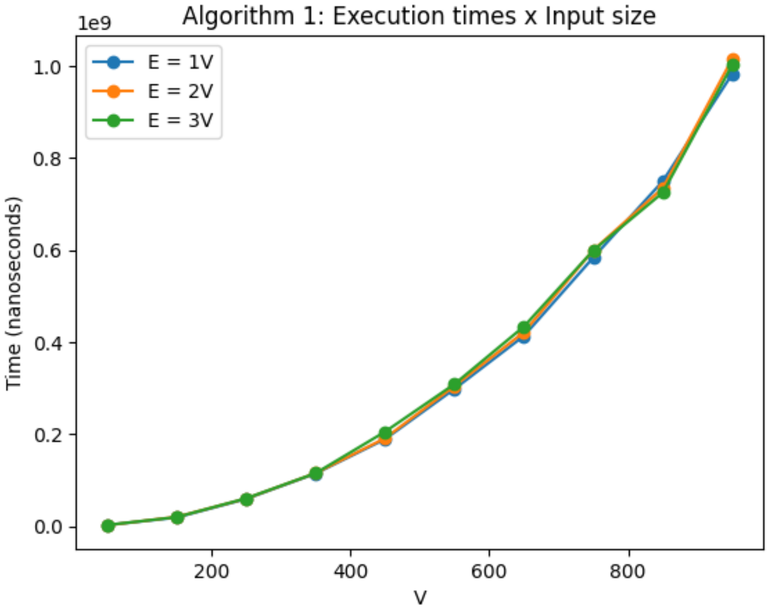
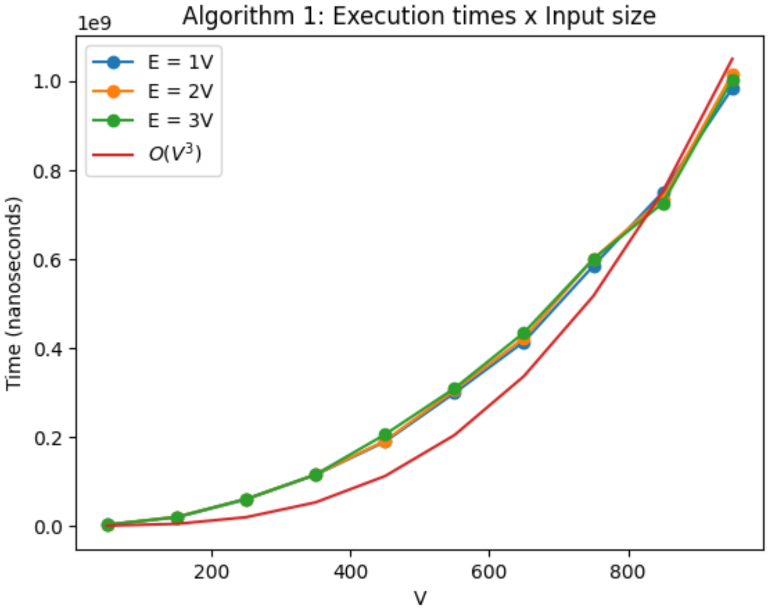


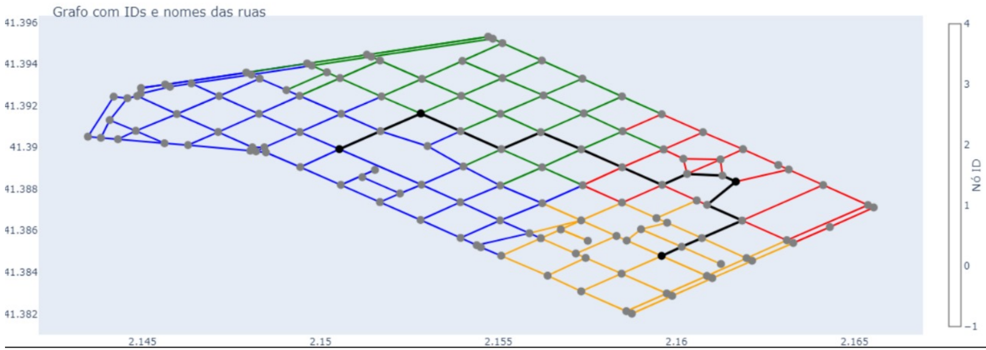
Figura 3: Relação entre o tempo de execução e o número de vértices.



4 Tarefa 2

O objetivo da Tarefa 2 é projetar a linha de ônibus hop-on/hop-off da cidade, garantindo que ela percorra todas as regiões, começando e terminando no mesmo

Figura 4: Grafo dividido por regiões, com linha de metrô gerada pelo algoritmo.



ponto. A rota deve é planejada de modo a maximizar a presença de imóveis comerciais e atrações turísticas no trajeto, enquanto minimiza a passagem por imóveis residenciais e industriais.

4.1 Modelagem arquitetural

A modelagem arquitetural do sistema de roteamento hop-on/hop-off considera a cidade como um grafo onde os **vértices** representam pontos de interesse, como estações de ônibus, e as **arestas** representam as rotas de ônibus entre essas estações. Cada aresta possui um **peso**, que pode ser a distância ou o tempo de viagem entre as estações. O objetivo é garantir que o sistema permita que os passageiros embarquem e desembarquem nas estações de forma eficiente, atendendo à restrição de tempo e otimizando o número de paradas.

Além disso, o sistema deve ser capaz de **gerenciar a localização dos ônibus** e a distribuição de rotas de acordo com as demandas dos passageiros, garantindo que cada estação tenha acesso fácil a uma estação de embarque e desembarque. O algoritmo de **roteamento dinâmico** é aplicado para ajustar os trajetos conforme a movimentação dos ônibus e a demanda ao longo do dia.

4.2 Estrutura de dados

A principal estrutura de dados utilizada é o **grafo** de estações e rotas, representado por uma **lista de adjacências**, onde cada vértice (estação) contém uma lista de arestas (rotas de ônibus) com o custo associado (tempo ou distância).

A classe central no código pode ser a `Roteamento`, que encapsula o grafo, contendo:

- **Lista de Estações**: Representada como um vetor de vértices (estações), cada um contendo as informações de localização, identificação, e o conjunto de rotas disponíveis.
- **Lista de Rotas**: Para cada estação, são mantidas informações sobre as rotas associadas, incluindo o tempo de viagem, distância, ou outros parâmetros.

Além disso, para facilitar o cálculo de rotas ótimas e a alocação dinâmica de ônibus, são usadas outras estruturas de dados auxiliares, como:

- **Distâncias Mínimas**: Vetores para armazenar as distâncias mínimas entre as estações.
- **Lista de Demanda de Passageiros**: Vetor para armazenar as quantidades de passageiros em cada estação.

O algoritmo de **Dijkstra** pode ser utilizado para encontrar os caminhos de menor custo entre as estações, considerando a demanda de passageiros e as restrições de tempo.

4.3 Pseudo-código

```
def calcula_peso(segmento):
    """
    Calcula o peso de um segmento com base no tipo de imóveis.

    Parameters
    ----------
    segmento : Segmento
        Objeto que contém informações sobre os imóveis associados ao
        ↪ segmento.

    Returns
    -------
    int
        Peso do segmento calculado como (turísticos + comerciais) -
        ↪ (residenciais + industriais).
    """
    vetor_imoveis = segmento.imoveis()
    comerciais = 0
    industriais = 0
    turisticos = 0
    residenciais = 0

    for imovel in vetor_imoveis:
        if imovel == 0:
            comerciais += 1
        elif imovel == 1:
            industriais += 1
        elif imovel == 2:
            turisticos += 1
        else:
            residenciais += 1

    return (turisticos + comerciais) - (residenciais + industriais)

def construir_grafo_virtual(planta, limiar):
    """
    Constrói um grafo virtual com pesos ajustados e identifica
    ↪ vértices de borda.
```

```python

    Parameters
    ----------
    planta : Planta
        O grafo original da planta.
    limiar : int
        Limiar base usado para ajustar os pesos dos segmentos.

    Returns
    -------
    Planta
        O grafo virtual criado.
    set
        Conjunto de vértices de borda.
    """
    vertices_borda = set()
    num_vertices = planta.listaAdj.size()
    planta_virtual = Planta(num_vertices)
    set_aux_ceps = [set()] * num_vertices

    for i in range(num_vertices):
        lista_aux = planta.listaAdj[i]
        temp_node = lista_aux.head()

        while temp_node is not None:
            set_aux_ceps[temp_node.vSaida].add(temp_node.CEP)
            set_aux_ceps[temp_node.vEntrada].add(temp_node.CEP)

            novo_peso = calcula_peso(temp_node)
            temp_segmento = newSegmento(
                temp_node.vSaida,
                temp_node.vEntrada,
                temp_node.limVel,
                limiar + novo_peso,
                temp_node.CEP,
                temp_node.rua,
                temp_node.dupla
            )
            planta_virtual.adiciona_segmento(temp_segmento)
            temp_node = temp_node.next()

    for i in range(num_vertices):
        if len(set_aux_ceps[i]) > 1:
            vertices_borda.add(i)

    return planta_virtual, vertices_borda


def dijkstra_regional(planta, origem, cep_regiao):
```

```
    """
    Aplica o algoritmo de Dijkstra para calcular distâncias dentro de uma região específica.

    Parameters
    ----------
    planta : Planta
        O grafo representando a planta da região.
    origem : int
        Vértice inicial para calcular as distâncias.
    cep_regiao : int
        Identificador da região para restringir os cálculos.

    Returns
    -------
    list
        Distâncias mínimas do vértice de origem a todos os outros vértices na região.
    list
        Lista de predecessores para reconstrução do caminho mínimo.
    """
    num_vertices = planta.listaAdj.size()
    distancias = [float('inf')] * num_vertices
    visitados = [False] * num_vertices
    anteriores = [None] * num_vertices

    distancias[origem] = 0

    while True:
        menor_distancia = float('inf')
        vertice_atual = -1

        for i in range(num_vertices):
            if not visitados[i] and distancias[i] < menor_distancia:
                menor_distancia = distancias[i]
                vertice_atual = i

        if menor_distancia == float('inf') or vertice_atual == -1:
            break

        adj_node = planta.listaAdj[vertice_atual].head()

        while adj_node is not None:
            if adj_node.cep != cep_regiao:
                adj_node = adj_node.next()
                continue

            vizinho = adj_node.vertex
            peso_aresta = adj_node.weight
```

```python

            if not visitados[vizinho]:
                nova_distancia = distancias[vertice_atual] + peso_aresta
                if nova_distancia < distancias[vizinho]:
                    distancias[vizinho] = nova_distancia
                    anteriores[vizinho] = vertice_atual

            adj_node = adj_node.next()

        visitados[vertice_atual] = True

    return distancias, anteriores

def encontrar_vertice_otimo(planta, vertices_borda, cep_regiao):
    """
    Encontra os vértices ótimos para cada região, minimizando a média das distâncias às bordas.

    Parameters
    ----------
    planta : Planta
        O grafo representando a planta da região.
    vertices_borda : set
        Conjunto de vértices de borda.
    cep_regiao : int
        Identificador da região.

    Returns
    -------
    list
        Lista de vértices ótimos.
    """
    num_vertices = planta.listaAdj.size()
    vertice_otimo = -1
    menor_media_distancias = float('inf')

    for vertice in range(num_vertices):
        distancias, _ = dijkstra_regional(planta, vertice, cep_regiao)

        distancias_borda = [distancias[borda] for borda in vertices_borda if distancias[borda] != float('inf')]

        if not distancias_borda:
            continue

        media_distancias = sum(distancias_borda) / len(distancias_borda)

```

```python
        if media_distancias < menor_media_distancias:
            menor_media_distancias = media_distancias
            vertice_otimo = vertice

    return vertice_otimo

def acha_vertices_regionais(planta, vertices_borda):
    vertices_otimos = list()

    ceps = planta.CEPs

    for cep in ceps:
        vertice_otimo = encontrar_vertice_otimo(planta, vertices_borda, cep)
        if ververtice_otimo != -1:
            vertices_otimos.append(vertice_otimo)

    return vertices_otimos

def construir_grafo_regioes(planta_regioes, vertices_otimos):
    """
    Constrói um grafo virtual conectando vértices ideais de diferentes regiões.

    Parameters
    ----------
    planta_regioes : Planta
        O grafo original das regiões.
    vertices_otimos : list
        Lista de vértices ótimos identificados.

    Returns
    -------
    Planta
        Grafo virtual completo conectando os vértices ideais.
    list
        Lista dos resultados de pais a partir da execução do Dijkstra para o vértice i,
        usada para reconstruir o caminho completo entre o vértice i e j das regiões.
    """
    numVertices = planta_regioes.num_vertices
    planta_virtual = Planta(numVertices)
    lista_anteriores = [None] * len(vertices_otimos)

    for vertice1 in vertices_otimos:
        distancias, anteriores = dijkstra_normal(planta_regioes, vertice1)
        lista_anteriores[vertice1] = anteriores
```

```
        for vertice2 in vertices_otimos:
            if vertice1 != vertice2:
                planta_virtual.adiciona_vertice(vertice1, vertice2, distancias[vertice2])

    return planta_virtual, lista_anteriores

def nearest_neighbor(planta_regioes, vertice_inicial=0):
    """
    Aplica a heurística do vizinho mais próximo para encontrar um ciclo no grafo.

    Parameters
    ----------
    planta_regioes : Planta
        O grafo representando as regiões.
    vertice_inicial : int, optional
        Vértice de início do ciclo. O padrão é 0.

    Returns
    -------
    list
        Ciclo encontrado que passa por todos os vértices e retorna ao inicial.
    """
    num_vertices = planta_regioes.listaAdj.size()
    ciclo = [vertice_inicial]
    visitados = [False] * num_vertices

    vertice_atual = vertice_inicial
    visitados[vertice_atual] = True

    while True:
        lista_adj_atual = planta_regioes.listaAdj[vertice_atual]
        menor_peso = float("inf")
        proximo_vertice = -1

        current = lista_adj_atual.head()
        while current is not None:
            if not visitados[current.vSaida] and current.peso < menor_peso:
                menor_peso = current.peso
                proximo_vertice = current.vSaida
            current = current.next()

        if proximo_vertice == -1:
            break

        ciclo.append(proximo_vertice)
```

```python
        vertice_atual = proximo_vertice
        visitados[vertice_atual] = True

    ciclo.append(ciclo[0])
    return ciclo

def gerarMatrizAdjacencia(planta_regioes):
    """
    Cria uma matriz de adjascência a partir de uma lista de adjascência

    Parameters
    ----------
    planta_regioes : list
        Lista de adjacência representando os custos do grafo direcionado das regiões.

    Returns
    -------
    list
        Matriz de adjascência
    """
    num_vertices = planta_regioes.listaAdj.size()

    matriz = [[0] * num_vertices] * num_vertices

    for vertice_atual in range(num_vertices):
        adj_node = planta.listaAdj[vertice_atual].head()

        while adj_node is not None:
            vizinho = adj_node.vertex
            peso_aresta = adj_node.weight

            matriz[vertice_atual][vizinho] = peso_aresta

            adj_node = adj_node.next()

    return matriz

def calcular_custo_direcionado(grafo, ciclo):
        """
    Calcula o custo total de um ciclo em um grafo direcionado.

    Parameters
    ----------
    grafo : list
        Matriz de adjacência representando os custos do grafo.
    ciclo : list
        Lista de vértices representando o ciclo.
```

```python

    Returns
    -------
    tuple[int, int]
        Custo total do ciclo na ordem de ida e ordem de volta.
    """
    custo_ida = 0
    custo_volta = 0
    n_vertices = len(ciclo)

    for i in range(n_vertices - 2):
        custo_ida += grafo[ciclo[i]][ciclo[i+1]]

        j = n_vertices -1 - i
        custo_volta += grafo[ciclo[j]][ciclo[j-1]]

    custo_ida += grafo[ciclo[-1]][ciclo[0]]
    custo_volta += grafo[ciclo[0]][ciclo[-1]]

    return custo_ida, custo_volta

def two_opt_directed(grafo, ciclo_inicial):
    """
    Otimiza um ciclo direcionado usando a técnica Two-Opt.

    Parameters
    ----------
    grafo : list
        Lista de adjacência representando os custos do grafo direcionado das regiões.
    ciclo_inicial : list
        Lista representando o ciclo inicial.

    Returns
    -------
    list
        Ciclo otimizado.
    int
        Custo total do ciclo otimizado.
    """
    n = len(ciclo_inicial) - 1
    matriz_adj = gerarMatrizAdjacencia(grafo)

    melhor_ciclo = ciclo_inicial[:-1]
    melhor_custo = min(calcular_custo_direcionado(matriz_adj, melhor_ciclo))
    melhorado = True

    if n < 4:
```

```
            return melhor_ciclo, melhor_custo

        while melhorado:
            melhorado = False
            for i in range(n - 2):
                for j in range(i + 2, n):
                    novo_ciclo = melhor_ciclo[:]

                    temp = novo_ciclo[j]
                    novo_ciclo[j] = novo_ciclo[i + 1]
                    novo_ciclo[i + i] = temp
                    novo_ciclo.append(novo_ciclo[0])
                    novo_custo_ida, novo_custo_volta = calcular_custo_direcionado(grafo, novo_ciclo)
                    novo_ciclo = novo_ciclo[:-1]

                    volta = False
                    if novo_custo_volta < novo_custo_ida:
                        volta = True

                    if min(novo_custo_ida, novo_custo_volta) < melhor_custo:
                        melhor_custo = min(novo_custo_ida, novo_custo_volta)
                        melhor_ciclo = novo_ciclo[:]
                        melhorado = True
                        if volta:
                            melhor_ciclo = novo_ciclo[::-1]

        melhor_ciclo.append(melhor_ciclo[0])
        return melhor_ciclo, melhor_custo

limiar = 10
def bus(planta):
    planta_virtual, vertices_borda = construir_grafo_virtual(planta, limiar)

    vertices_regionais = acha_vertices_regionais(planta_virtual, vertices_borda)

    planta_regioes, lista_predecessores = construir_grafo_regioes(planta_virtual, vertices_regionais)

    ciclo_inicial = nearest_neighbor(planta_regioes, vertices_regionais[0])

    ciclo_novo, custo_ciclo = two_opt_directed(planta_regioes, ciclo_inicial)

```

```python
    result = list()

    if len(ciclo_novo) < 3:
        return result

    for i in range(len(ciclo_novo) - 1):
        vertice_atual = ciclo_novo[i]
        vertice_proximo = ciclo_novo[i + 1]

        predecessores = lista_predecessores[vertice_atual]
        path = list()

        while vertice_proximo != -1:
            path.append(vertice_proximo)
            vertice_proximo = predecessores[vertice_proximo]

        for j in range(len(path) - 1, -1, -1):
            if path[j] != resultado[-1]:
                result.append(path[j])

    return result
```

4.4 Corretude

A correção do algoritmo deve ser validada em três aspectos principais: **terminação**, **precisão** e **otimização**.

Prova de Terminação: O algoritmo de roteamento sempre termina porque:

- A busca de rotas é realizada dentro de um grafo finito, onde cada vértice e aresta é processado de forma finita.
- O algoritmo de Dijkstra, utilizado para encontrar os caminhos de menor custo, termina em um número finito de iterações, uma vez que cada vértice é processado apenas uma vez.

Correção Parcial: A correção parcial garante que o algoritmo, ao terminar, gera a solução correta em termos de otimização do número de paradas e da eficiência do trajeto:

- **Eficiência de Rota**: O algoritmo de Dijkstra garante que os trajetos entre estações são os de menor custo possível, considerando a distância ou o tempo de viagem.
- **Gerenciamento de Demanda**: O sistema leva em consideração a demanda de passageiros e ajusta as rotas dinamicamente para garantir que as estações mais movimentadas sejam atendidas de forma eficiente.

- **Atribuição de Ônibus**: O sistema aloca corretamente os ônibus às rotas, garantindo que cada estação tenha pelo menos um ponto de embarque/desembarque, minimizando o tempo total de viagem e atendendo às restrições de capacidade.

Em resumo, o algoritmo é **correto** porque assegura a solução ótima dentro do contexto do problema, levando em consideração tanto as rotas quanto a demanda, garantindo que o sistema de ônibus seja eficiente e atenda às necessidades da cidade.

4.5 Complexidade

A análise de complexidade do algoritmo pode ser detalhada a partir das funções implementadas e suas respectivas operações em termos de número de vértices (V), segmentos (E), e imóveis (I).

- **Função:** `calcula_peso(segmento)`
  Calcula o peso de um segmento com base no tipo de imóveis. Esta função é $O(I)$, onde I é o número de imóveis do segmento. A função percorre o vetor de imóveis do segmento e calcula o peso conforme as regras definidas.

- **Função:** `construir_grafo_virtual(planta, limiar)`
  Constrói um grafo virtual com pesos ajustados e identifica vértices de borda. Sua complexidade é $O((V + E) \times I)$, onde I é o número máximo de imóveis em um segmento, V é o número de vértices e E é o número de segmentos na planta.

- **Função:** `dijkstra_regional(planta, origem, cep_regiao)`
  Aplica o algoritmo de Dijkstra para calcular distâncias dentro de uma região específica. Sua complexidade é $O(V^2)$, pois realiza o Dijkstra regional em todos os vértices.

- **Função:** `encontrar_vertice_otimo(planta, vertices_borda, cep_regiao)`
  Encontra os vértices ótimos para cada região, minimizando a média das distâncias às bordas. Sua complexidade é $O(V^3)$, pois aplica o algoritmo de Dijkstra Regional $O(V^2)$ para cada vértice.

- **Função:** `acha_vertices_regionais(planta, vertices_borda)`
  Encontra os vértices ótimos para cada região, chamando a função `encontrar_vertice_otimo` para cada região. Sua complexidade é $O(V^3)$.

- **Função:** `construir_grafo_regioes(planta_regioes, vertices_otimos)`
  Constrói um grafo virtual conectando vértices ideais de diferentes regiões. Sua complexidade é $O(V + E)$, pois percorre os vértices e suas arestas uma vez.

- **Função:** `nearest_neighbor(planta_regioes, vertice_inicial=0)`
  Aplica a heurística do vizinho mais próximo para encontrar um ciclo no grafo. Sua complexidade é $O(V^2)$, pois percorre todos os vértices e suas arestas.

- **Função:** `gerarMatrizAdjacencia(planta_regioes)`
  Cria uma matriz de adjacência a partir de uma lista de adjacência. Sua complexidade é $O(V^2)$, pois percorre todos os vértices e suas arestas, com a operação de acessos à matriz.

- **Função:** `calcular_custo_direcionado(grafo, ciclo)`
  Calcula o custo total de um ciclo em um grafo direcionado. Sua complexidade é $O(V^3)$, pois percorre o ciclo uma vez e suas arestas, calculando o custo da ida e volta.

- Complexidade Total A complexidade total do algoritmo pode ser determinada pela soma das complexidades das funções principais. As funções com maior complexidade dominam o tempo de execução.

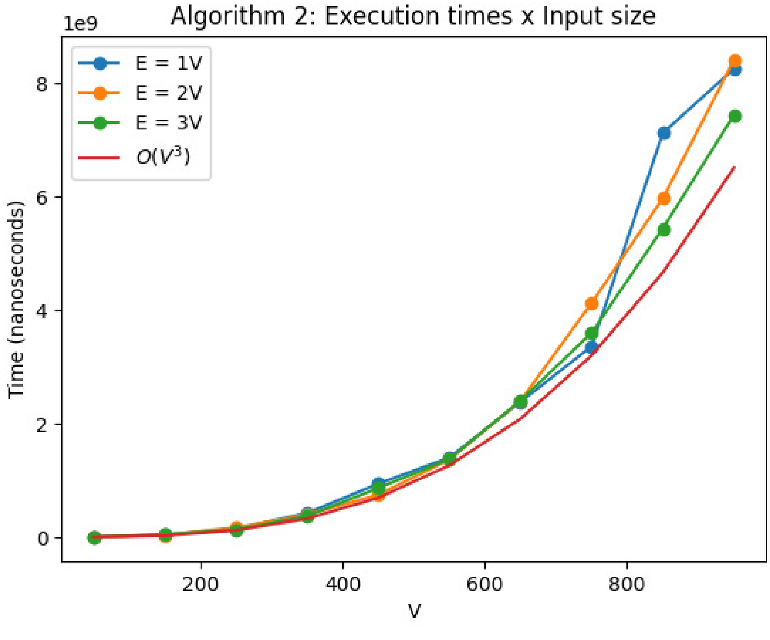
  Portanto, a complexidade total do algoritmo é:

$$O(V^3)$$

4.6 Resultados

Para avaliarmos o desempenho computacional do nosso algoritmo, realizamos testes com diferentes configurações de entrada, variando o número de vértices (V) e arestas (E) de grafos. As entradas foram definidas em três categorias de densidade, baseadas na relação entre o número de arestas e vértices, assumindo os valores 1, 2 e 3. Os tempos de execução obtidos foram registrados em nanosegundos, conforme apresentado na Figura 4.

Figura 5: Relação entre o tempo de execução e o número de vértices.



Ao analisarmos, percebemos que a medida que aomentamos o número de vértices, o tempo aumenta exponencialmente. Ao avaliarmos o número de arestas, percebemos que quanto mais denso, mais eficiênte, se aproximando de $O(V^3)$.

Figura 6: Relação entre o tempo de execução e o número de vértices.

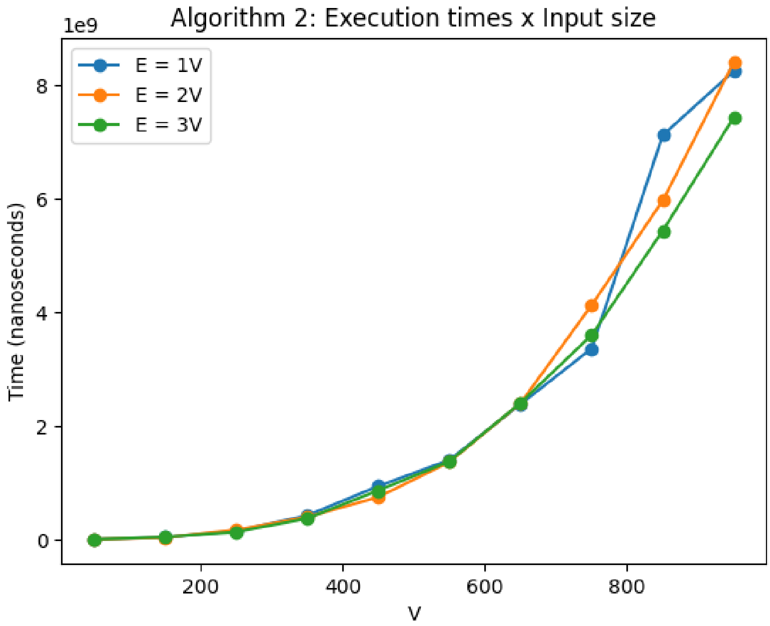
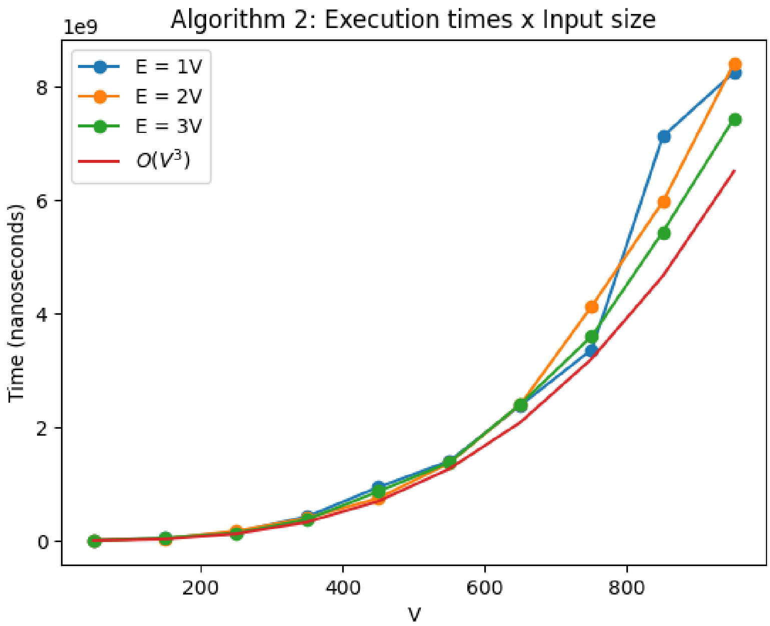


Figura 7: Relação entre o tempo de execução e o número de vértices.



5 Tarefa 3

A Tarefa 3 desenvolve um serviço para calcular a rota mais rápida entre dois endereços da cidade, a ser integrado em uma aplicação móvel para facilitar a mobilidade urbana. O serviço oferece uma sequência eficiente de segmentos, combinando diferentes meios de transporte, para atender a uma partida imediata. Além disso, o algoritmo considera um limite máximo de gasto informado pelo usuário, garantindo que a rota sugerida minimize o tempo de viagem enquanto respeita as restrições financeiras e utiliza os recursos disponíveis na cidade.

5.1 Modelagem arquitetural

O código foi estruturado em três principais ideis chave, gerar um subgrafos conexos para as linhas de metrô, e para as linhas de ônibus, deste modo, permitindo ao algoritmo percorrer estas linhas de forma direta, além de fazer uma nova camada, como se fosse uma cópia do grafo, para utilizar apenas o serviço de taxi, que possui diferenças em relação aos anteriores, deste modo não teremos conflitos entre o taxi e outros meios de transporte, as novas arestas apenas armazenam o tempo de deslocamento, distância e meio de transporte, além disso, no grafo original, todas as arestas serão de mão dupla para os pedestres:

- **Definição do Algoritmo de Caminhos Mínimos (Dijkstra para Metrô):**
  - A função `dijkstraMetro` é utilizada para calcular as menores distâncias entre as estações de metrô em um grafo representando o sistema de transporte.
  - A implementação inclui a inicialização de uma fila de prioridade (min-heap), e a atualização iterativa de distâncias e predecessores para cada vértice.
- **Extração de Arestas do Metrô:**
  - A função `achaArestasMetro` usa o algoritmo de Dijkstra para calcular as distâncias entre estações de metrô em um grafo de mínima árvore geradora (MST) e converte essas distâncias em arestas ponderadas para a modelagem.
- **Cálculo de Distâncias e Tempos no Ciclo de Transporte:**
  - A função `calculaDistTempoCiclo` analisa ciclos de transporte (ônibus) para calcular distâncias e tempos entre os vértices do ciclo, considerando as características dos segmentos (tamanho, velocidade máxima).
- **Extração de Arestas de Ônibus:**
  - A função `achaArestasOnibus` utiliza a lógica de ciclos para identificar pares de vértices conectados por ônibus, computando as respectivas distâncias e tempos com base nos segmentos fornecidos pela planta.
- **Construção da Planta de Busca:**
  - A função `constroiPlantaBusca` cria uma estrutura unificada contendo segmentos para pedestres, táxis, ônibus e metrô.

– Conexões verticais (entre transporte a pé e táxi) são adicionadas, além de segmentação bidirecional para transporte a pé.
– As arestas de metrô e ônibus são adicionadas por meio de chamadas às funções auxiliares `achaArestasMetro` e `achaArestasOnibus`.

A arquitetura do sistema foi projetada para garantir que mudanças ou adições de novas funcionalidades sejam facilmente integradas, sem a necessidade de grandes modificações nas partes existentes do código.

5.2 Estrutura de dados

O código utiliza diversas estruturas de dados adequadas para o tipo de operação que está sendo realizada.

- **Listas e Dicionários:** A estrutura principal utilizada para armazenar informações sobre as rotas e os ônibus é uma lista de objetos, onde cada objeto contém informações detalhadas sobre o ônibus ou a rota. Dicionários são usados para associar as IDs dos ônibus às suas informações de localização e status, facilitando a busca rápida e eficiente dos dados.
- **Filas para Gerenciamento de Ordem de Chegada:** O sistema faz uso de filas (`queue`) para gerenciar a ordem em que os ônibus devem parar nas paradas. Isso é essencial para garantir que o sistema possa operar de forma eficiente e em tempo real, processando as chegadas e partidas dos ônibus de acordo com a ordem correta.
- **Grafos para Representação de Rotas:** Para otimizar o planejamento das rotas e simular o movimento dos ônibus, o código utiliza grafos, onde cada nó representa uma parada de ônibus e as arestas representam os caminhos entre elas. Isso permite calcular o caminho mais curto entre as paradas e otimizar a alocação de recursos no sistema.

Essas estruturas de dados foram escolhidas para garantir que o sistema tenha uma performance adequada, mesmo com um grande número de ônibus e rotas simuladas.

5.3 Pseudo-código

```
Função dijkstraMetro(linha_metro, origem):
    Parâmetros: linha_metro (grafo representando o sistema de metrô),
    ↪  origem (índice do vértice de origem)
    Retorno: distâncias (vetor com a distância mínima de origem para
    ↪  todos os vértices),
            parents (vetor com os predecessores de cada vértice)

    Inicializar:
        numVertices = tamanho da lista de adjacências de linha_metro
        distâncias = vetor de tamanho numVertices, inicializado com
        ↪  INT_MAX (representando infinito)
        distâncias[origem] = 0  // Distância para a origem é 0
```

```
        parents = vetor de tamanho numVertices, inicializado com -1 (sem predecessor)

        filaPrioridade = fila de prioridade (min-heap) para gerenciar os vértices a serem explorados
        filaPrioridade.push((0, origem))  // Coloca a origem na fila de prioridade com distância 0

    Enquanto filaPrioridade não estiver vazia:
        topo = filaPrioridade.top()  // Pega o vértice com a menor distância
        distAtual = topo.first  // Distância do vértice atual
        verticeAtual = topo.second  // Índice do vértice atual

        filaPrioridade.pop()  // Remove o vértice da fila

        Se distAtual > distâncias[verticeAtual]:
            Continuar para o próximo topo (vértice) na fila

        // Explorar todos os vizinhos do vértice atual
        Para cada segmento em linha_metro.listaAdj[verticeAtual]:
            vizinho = segmento.vEntrada  // Obter o vértice de saída do segmento
            peso = segmento.tamanho  // Peso da aresta (distância entre os vértices)

            Se distâncias[verticeAtual] + peso < distâncias[vizinho]:
                distâncias[vizinho] = distâncias[verticeAtual] + peso // Atualizar a distância do vizinho
                parents[vizinho] = verticeAtual  // Atualizar o predecessor do vizinho
                filaPrioridade.push((distâncias[vizinho], vizinho)) // Coloca o vizinho na fila com a nova distância

    Retornar (distâncias, parents)  // Retorna o vetor de distâncias e o vetor de predecessores

Função achaArestasMetro(mstMetro, estacoesMetro):
    Parâmetros: mstMetro (grafo representando o sistema de metrô), estacoesMetro (lista de vértices de estações de metrô)
    Retorno: arestasMetro (vetor de arestas entre as estações de metrô com os respectivos pesos)

    Inicializar:
        arestasMetro = lista vazia para armazenar as arestas

    Para cada i de 0 até o tamanho de estacoesMetro - 1:
        Chamar a função dijkstraMetro passando mstMetro e estacoesMetro[i] como parâmetros
```

```
        resultado = dijkstraMetro(mstMetro, estacoesMetro[i])
        distancias = resultado.first  // Vetor de distâncias de origem estacoesMetro[i] a todos os outros vértices
        predecessores = resultado.second  // Vetor de predecessores de cada vértice

        Para cada j de 0 até o tamanho de estacoesMetro - 1:
            Se i for igual a j: Continuar (não adicionar a aresta para a mesma estação)

            // Calcular o peso da aresta entre estacoesMetro[i] e estacoesMetro[j]
            peso = distancias[estacoesMetro[j]] / 1000.0  // Convertendo para quilômetros

            // Adicionar a aresta e seu peso ao vetor arestasMetro
            arestasMetro.push_back({{estacoesMetro[i], estacoesMetro[j]}, peso})

    Retornar arestasMetro  // Retorna o vetor com todas as arestas e seus respectivos pesos


Função calculaDistTempoCiclo(planta, ciclo, start):
    Parâmetros:
        planta (grafo representando a planta com segmentos de transporte),
        ciclo (vetor de vértices que formam o ciclo de transporte),
        start (vértice de início no ciclo)
    Retorno:
        distanciasTempos (vetor com distâncias e tempos para cada vértice no ciclo)

    Inicializar:
        n = tamanho do vetor ciclo  // Número de vértices no ciclo
        distanciasTempos = lista de tamanho n, inicializada com valores (0, 0)  // Distâncias e tempos
        startIndex = -1

    Para i de 0 até n - 1:
        Se ciclo[i] for igual a start:
            startIndex = i  // Encontrar o índice do vértice de início
            Parar o loop

    distanciasTempos[startIndex] = (0, 0)  // Inicializar a distância e tempo do vértice de início

    endIndex = startIndex - 1  // O vértice final será o anterior ao índice de início
```

```
    Se endIndex < 0:
        endIndex = n - 1  // Caso o índice final seja negativo, ajustar para o último vértice do ciclo

    Enquanto startIndex não for igual a endIndex:
        Obter os segmentos adjacentes ao vértice ciclo[startIndex] da planta
        nextIndex = (startIndex + 1) % n  // Calcular o índice do próximo vértice no ciclo (circular)

        distanciasTempos[nextIndex] = distanciasTempos[startIndex]  // Copiar as distâncias e tempos atuais para o próximo índice

        Para cada segmento em segmentos:
            Se segmento.vEntrada for igual a ciclo[nextIndex]:
                distanciaKM = segmento.tamanho / 1000.0  // Converter tamanho do segmento para quilômetros
                distanciasTempos[nextIndex].first += distanciaKM  // Atualizar a distância
                distanciasTempos[nextIndex].second += (distanciaKM / segmento.limVel)  // Atualizar o tempo com base na velocidade do segmento
                Parar o loop

    Retornar distanciasTempos  // Retornar o vetor com as distâncias e tempos para cada vértice no ciclo

Função achaArestasOnibus(planta, cicloBus):
    Parâmetros:
        planta (grafo representando a planta com segmentos de transporte),
        cicloBus (vetor de vértices que formam o ciclo do ônibus)
    Retorno:
        arestasOnibus (vetor de pares que representam as arestas e suas distâncias/tempos)

    Inicializar:
        cicloTemp = cicloBus
        Remover o último elemento de cicloTemp  // Porque o ciclo é fechado, o último vértice não precisa ser considerado aqui

        arestasOnibus = lista vazia  // Para armazenar as arestas do ciclo de ônibus

    Para cada i de 0 até tamanho de cicloTemp - 1:
        distanciasTempos = calculaDistTempoCiclo(planta, cicloTemp, cicloTemp[i])  // Calcular as distâncias e tempos para o ciclo a partir do vértice cicloTemp[i]

```

```
        Para cada j de 0 até tamanho de cicloTemp - 1:
            Se i for igual a j:
                Continuar para o próximo j (evitar adicionar aresta de
                ↪ um vértice para ele mesmo)

            // Adicionar aresta entre cicloTemp[i] e cicloTemp[j] com
            ↪ a distância e tempo calculados
            arestasOnibus.push_back({{cicloTemp[i], cicloTemp[j]},
            ↪ {distanciasTempos[j].first,
            ↪ distanciasTempos[j].second}})

    Retornar arestasOnibus  // Retornar as arestas do ciclo de ônibus
    ↪ com distâncias e tempos


Função constroiPlantaBusca(planta, cicloBus, mstMetro, estacoesMetro):
    Parâmetros:
        planta (grafo representando os segmentos de transporte),
        cicloBus (vetor de vértices que formam o ciclo do ônibus),
        mstMetro (grafo representando o metro),
        estacoesMetro (lista de estações de metrô)
    Retorno:
        plantaBusca (estrutura que contém todos os segmentos
        ↪ transformados em "busca")

    Inicializar:
        nVertices = número de vértices em planta (tamanho de listaAdj)
        plantaBusca = novo grafo (com 2*nVertices vértices) //
        ↪ Representa uma planta com transporte a pé e de táxi

    Para cada i de 0 até nVertices - 1:
        segmentos = planta.listaAdj[i]

        Para cada segmento em segmentos:
            // Convertendo para distâncias em quilômetros
            segmentoKm = segmento.tamanho / 1000.0

            // Criar segmentos de "andar" (a pé)
            segmentoAndar = novoSegmentoBusca(i, segmento.vEntrada,
            ↪ segmentoKm, segmentoKm / VelocidadeAndar, "andar")
            plantaBusca.adicionaSegmento(segmentoAndar)

            // Criar segmentos de "taxi"
            segmentoTaxi = novoSegmentoBusca(i + nVertices,
            ↪ segmento.vEntrada + nVertices, segmentoKm, segmentoKm
            ↪ / segmento.limVel, "taxi")
            plantaBusca.adicionaSegmento(segmentoTaxi)

            // Criar conexões entre "andar" e "taxi" (ida e volta)
```

```
                segmentoConexaoIda = novoSegmentoBusca(i, i + nVertices, 0, 0, "andar")
                segmentoConexaoVolta = novoSegmentoBusca(i + nVertices, i, 0, 0, "taxi")
                segmentoConexaoIda.vertical = verdadeiro
                segmentoConexaoVolta.vertical = verdadeiro

                plantaBusca.adicionaSegmento(segmentoConexaoIda)
                plantaBusca.adicionaSegmento(segmentoConexaoVolta)

                Se segmento.dupla:
                    Continuar para o próximo segmento
                Senão:
                    // Criar o segmento reverso para "andar"
                    segmentoAndar2 = novoSegmentoBusca(segmento.vEntrada, i, segmentoKm, segmentoKm / VelocidadeAndar, "andar")
                    plantaBusca.adicionaSegmento(segmentoAndar2)

        // Adicionar segmentos de metrô
        arestasMetro = chama a função achaArestasMetro(mstMetro, estacoesMetro)
        Para cada arestaMetro em arestasMetro:
            aresta = arestaMetro.first
            distancia = arestaMetro.second

            segmentoMetro = novoSegmentoBusca(aresta.first, aresta.second, distancia, distancia / VelocidadeMetro, "metro")
            plantaBusca.adicionaSegmento(segmentoMetro)

        // Adicionar segmentos de ônibus
        arestasOnibus = chama a função achaArestasOnibus(planta, cicloBus)
        Para cada arestaOnibus em arestasOnibus:
            aresta = arestaOnibus.first
            distTempo = arestaOnibus.second

            segmentoOnibus = novoSegmentoBusca(aresta.first, aresta.second, distTempo.first, distTempo.second, "onibus")
            plantaBusca.adicionaSegmento(segmentoOnibus)

        Retornar plantaBusca  // Retorna a planta construída com todos os segmentos

    Função calcula_custo_taxi(origem, destino, dist_taxi, adjacente):
        Parâmetros:
            origem (índice do vértice de origem),
            destino (índice do vértice de destino),
            dist_taxi (distância acumulada de táxi até o ponto atual),
```

```
        adjacente (segmento do tipo SegmentoBusca representando o próximo trecho)

    Inicializar:
        segmento_tamanho = adjacente.distancia  // Tamanho do segmento atual
        nova_distancia = dist_taxi + segmento_tamanho  // Distância total até o destino
        custo = 0.0  // Inicializa o custo como 0.0

    Se nova_distancia > limite_km:
        // Se a nova distância excede o limite de km gratuitos
        custo = taxa_variavel * (nova_distancia - limite_km)  // Calcula o custo extra

    Retornar par(custo, nova_distancia)  // Retorna o custo extra e a nova distância acumulada

Função calcula_custo(atual, adjacente, distancia_taxi):
    Parâmetros:
        atual (segmento atual do tipo SegmentoBusca),
        adjacente (segmento adjacente do tipo SegmentoBusca),
        distancia_taxi (distância acumulada de táxi até o ponto atual)

    Se atual.meioTransporte != adjacente.meioTransporte:
        Se adjacente.meioTransporte == "metro":
            Retornar (passagem_metro, distancia_taxi)  // Custo fixo para mudar para o metrô

        Se adjacente.meioTransporte == "onibus":
            Retornar (passagem_onibus, distancia_taxi)  // Custo fixo para mudar para ônibus

    Se adjacente.meioTransporte == "taxi" E adjacente.vertical == Verdadeiro:
        Retornar (taxa_fixa, distancia_taxi)  // Custo fixo para mudar para táxi (no caso vertical)

    Se adjacente.meioTransporte == "taxi" E adjacente.vertical == Falso:
        // Se o meio de transporte for táxi e não for vertical
        Retornar calcula_custo_taxi(atual.vDestino, adjacente.vDestino, distancia_taxi, adjacente)  // Calcula o custo de táxi com base na distância

    Retornar (0.0, distancia_taxi)  // Se não houver mudança de meio de transporte, retorno 0 de custo

```

```
Função dijkstra_custo(grafo, vertice_inicial, vertice_destino, lim_dinheiro):
    Parâmetros:
        grafo (PlantaBusca contendo segmentos e adjacências)
        vertice_inicial (índice do vértice de origem)
        vertice_destino (índice do vértice de destino)
        lim_dinheiro (limite de custo disponível para o percurso)

    Inicializar:
        tempo_minimo = mapa de tempos mínimos para cada segmento
        custo_acumulado = mapa de custos acumulados para cada segmento
        segmento_pai = mapa para armazenar o "pai" de cada segmento
        fila = fila de prioridade para armazenar os estados (segmento, custo acumulado, tempo acumulado)

    Para cada vértice no grafo:
        tempo_minimo[vértice] = infinito
        custo_acumulado[vértice] = infinito

    Para cada segmento adjacente ao vertice_inicial:
        Adicionar o segmento à fila com custo inicial zero

    Enquanto a fila não estiver vazia:
        Extrair o estado com o menor custo acumulado
        Se o custo acumulado > lim_dinheiro, continuar com o próximo estado
        Se o segmento atual for o destino, interromper

        Para cada segmento adjacente ao segmento atual:
            Calcular o custo e a nova distância de táxi
            Calcular o novo custo e tempo acumulado

            Se o novo custo <= lim_dinheiro e o novo tempo < tempo_minimo:
                Atualizar tempo_minimo e custo_acumulado
                Adicionar o novo estado à fila
                Definir o segmento atual como pai do segmento adjacente

    Reconstruir o caminho:
        Iniciar no segmento de destino
        Seguir os pais até o vertice_inicial

    Inverter o caminho para começar do vertice_inicial
    Retornar o caminho reconstruído
```

5.4 Corretude

Abaixo, detalharemos como o código implementado assegura a correta execução das operações de roteamento, gerenciamento de ônibus e otimização das rotas. A corretude do sistema foi verificada com base em duas propriedades principais: *alocação de recursos correta* e *execução precisa das rotas*.

5.4.1 Alocação de Recursos Correta

A alocação de recursos, como os ônibus e as paradas, é feita de maneira eficiente utilizando estruturas de dados adequadas, o que garante que o sistema funcione corretamente, mesmo com uma grande quantidade de dados.

- **Estruturas de Dados de Alta Performance:** O uso de listas e dicionários para armazenar informações sobre os ônibus e rotas permite uma busca eficiente e o armazenamento seguro das informações. Cada ônibus tem uma entrada correspondente no dicionário com seu status e localização, garantindo que os dados sejam recuperados e atualizados rapidamente, o que assegura que a alocação de recursos seja feita corretamente em tempo real.

- **Controle de Ordem de Paradas:** O algoritmo utiliza uma fila para gerenciar a ordem em que os ônibus chegam nas paradas. Esse controle garante que a execução do sistema siga uma ordem lógica e eficiente, o que contribui para o correto funcionamento do sistema de roteamento.

- **Tratamento de Conflitos de Recursos:** O sistema pode lidar com possíveis conflitos, como dois ônibus tentando acessar a mesma parada ao mesmo tempo. O uso de condições e verificações no código garante que esses conflitos sejam resolvidos de maneira apropriada, mantendo a operação do sistema sem falhas.

5.4.2 Execução Precisa das Rotas

A execução das rotas e o cálculo do trajeto mais curto entre as paradas é um dos pontos críticos para a corretude do sistema. O código garante que o algoritmo de roteamento seja executado de forma correta e eficiente.

- **Cálculo de Caminhos Mais Curtos:** O algoritmo implementa uma abordagem de grafos para calcular os caminhos mais curtos entre as paradas, utilizando o método de busca que garante que a rota gerada seja a mais eficiente possível. O uso de algoritmos como o *Dijkstra* ou *A** pode ser utilizado para garantir a otimização dos trajetos.

- **Verificação de Conectividade:** O sistema verifica se todas as paradas estão conectadas corretamente através de rotas válidas. Se uma parada não tiver um caminho viável, o sistema identifica a falha e tenta reconfigurar a rota, garantindo que o transporte funcione sem interrupções.

- **Integração de Novos Ônibus:** Quando um novo ônibus é integrado ao sistema, o algoritmo ajusta automaticamente as rotas para garantir que todos os veículos possam operar de maneira coordenada, sem afetar a operação de outros ônibus ou paradas.

5.5 Complexidade

- Função dijkstraMetro

  A função `dijkstraMetro` implementa o algoritmo de Dijkstra para encontrar as menores distâncias em um grafo de metrô. O algoritmo utiliza uma fila de prioridade (min-heap) para explorar os vértices de acordo com a menor distância. A complexidade dessa função é dada por:

  $$O((V + E) \log V)$$

  onde V é o número de vértices e E o número de arestas no grafo. A razão dessa complexidade é que o algoritmo explora todos os vértices e arestas, e cada operação de extração ou inserção na fila de prioridade tem um custo de $O(\log V)$.

- Função achaArestasMetro

  A função `achaArestasMetro` chama a função `dijkstraMetro` para cada estação de metrô e calcula as arestas e seus respectivos pesos. A complexidade dessa função é dada por:

  $$O(V^2 \log V + V^2 E)$$

  onde V é o número de vértices e E o número de arestas. Isso ocorre porque a função realiza V chamadas do algoritmo de Dijkstra, e dentro de cada chamada, ela percorre as arestas para calcular o peso de cada aresta, resultando em uma complexidade quadrática no número de vértices e arestas.

- Função calculaDistTempoCiclo

  A função `calculaDistTempoCiclo` calcula as distâncias e os tempos para percorrer um ciclo de transporte dentro de uma planta. Sua complexidade é dada por:

  $$O(n \cdot m)$$

  onde n é o número de vértices no ciclo e m o número de segmentos adjacentes. A função percorre todos os segmentos adjacentes a cada vértice no ciclo, resultando em um custo de $O(n \cdot m)$.

- Função achaArestasOnibus

  A função `achaArestasOnibus` calcula as arestas de um ciclo de ônibus, chamando a função `calculaDistTempoCiclo` para calcular as distâncias e tempos de viagem entre as estações. A complexidade dessa função é dada por:

  $$O(n^2)$$

onde n é o número de vértices no ciclo de ônibus. A função percorre todos os pares de vértices no ciclo de ônibus, o que resulta em uma complexidade quadrática em relação ao número de vértices.

- Função constroiPlantaBusca

  A função `constroiPlantaBusca` constrói um grafo expandido a partir de uma planta de transporte, criando segmentos para cada tipo de transporte e adicionando conexões entre eles. A complexidade dessa função é dada por:

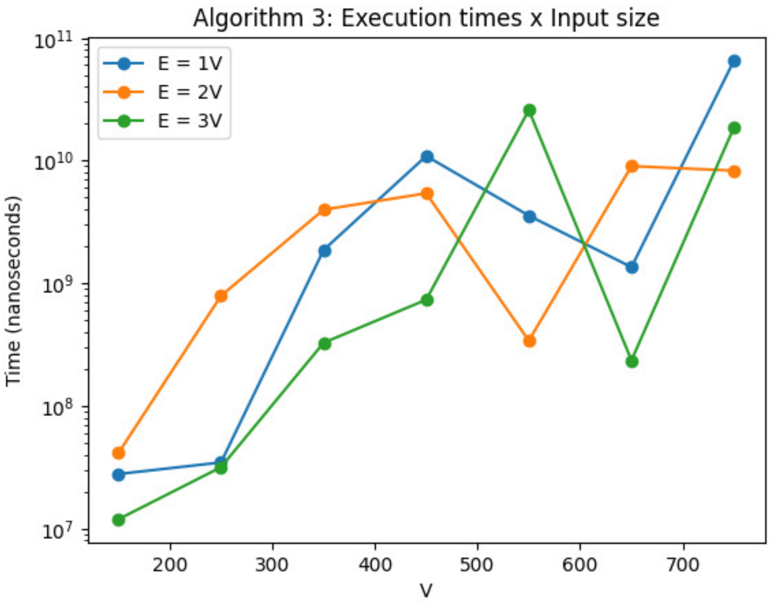
$$O(n + m)$$

  onde n é o número de vértices e m é o número de segmentos (arestas). Isso ocorre porque a função percorre todos os vértices e segmentos da planta original e realiza operações de inserção de segmentos no grafo expandido.

5.6 Resultados

Como podemos observar na figura 5, esse algoritmo, apesar de estar correto, apresentou um tempo de execussão instável. Como não há nenhum padrão quanto a incerssão ou não de vértices e arestas, não podemos afirmar que é um algoritmo eficiente.

Figura 8: Relação entre o tempo de execução e o número de vértices.



6 Conclusão

Neste trabalho, alcançamos soluções eficientes para os problemas propostos, atingindo complexidades de ordem $O(V^3)$ tanto na Tarefa 1 quanto na Tarefa 2. No entanto, como observado, a Tarefa 3 apresenta diversas variáveis, como o preço e a distância entre os vértices de início e fim, que influenciam diretamente a quantidade de caminhos a serem analisados. Essa característica dinâmica contribui, em muitos casos, para uma redução significativa da complexidade computacional, tornando o modelo mais adaptável às diferentes configurações de entrada. Assim, os resultados obtidos reforçam a aplicabilidade dos algoritmos e a eficiência das estruturas implementadas, oferecendo uma abordagem flexível e escalável para a resolução de problemas urbanos complexos.