Fundação Getúlio Vargas

Projeto e Análise de Algoritmos

Projeto A2

FGV EMAp

Paula Eduarda de Lima

Mariana Fernandes Rocha

Ana Júlia Amaro Pereira Rocha

Maria Eduarda Mesquita Magalhães

Ciência de Dados e Inteligência Artificial
4 º período

Conteúdo

Modelagem

1.1 O Grafo

A modelagem do problema em formato de grafo seguiu os seguintes princípio:

- O grafo não direcionado com pesos representa a planta de cidade Vargas.
- Cada aresta é um segmento da cidade.
- Os pesos representam a distância e, consequentemente, a quantidade de imóveis no segmento.
- Uma rua é um conjunto de aresta/seguimentos adjacentes.
- Cada vértice é um cruzamento.

1.2 Geração do Grafo com Python

1.2.1 Criação do Grafo Base

Foi feita uma função em Python que gera um grafo em forma de grade $n \times n$, representando a cidade com n^2 cruzamentos conectados por segmentos. Um conjunto de segmentos (consecutivos na horizontal ou vertical) é uma rua.

1.2.2 Divisão em Regiões

Utilizou-se o algoritmo de clusterização K-Means para dividir as ruas da cidade em 12 regiões.

1.2.3 Atribuição de Arestas

Cada rua (conjunto de arestas adjacentes) foi associada à região. Foram criadas ruas contínuas horizontais e verticais dentro de cada região, respeitando o comprimento mínimo e máximo (parâmetros L e K).

1.2.4 Atributos das Arestas

Adicionaram-se propriedades como:

- **Número da Rua:** Identificador único para cada rua.
- **CEP:** Número identificando a região.
- **Trânsito:** Um valor aleatório entre 0 e 1 para simular condições de tráfego.
- **Distância:** Valor aleatório (de 5 à 10) representando o comprimento da rua/número de imóveis.
- **V1 e V2:** Vértices das extremidades da aresta.
- **Id Edge:** Identificador único para cada aresta.

1.2.5 Atributos dos Vértices

Adicionaram-se propriedades como:

- **Id Vertex:** Identificador único para cada vértice.
- **Is Metro Station:** Para indicar se são estações de metrô (inicialmente falso para todos).

1.2.6 Conversão e Modificação para JSON

Formatação Node-Link (vértice-aresta): O grafo foi convertido para o formato Node-Link, comum para representar grafos em JSON.

1.2.7 Visualização do Grafo

Foi gerado um gráfico onde:

- **Vértices** são exibidos como círculos azuis.
- **Arestas** são coloridas conforme as regiões.
- **Números das ruas** foram adicionados para melhor identificação.

1.2.8 Exemplo de Grafo

A seguir está o exemplo de um dos grafos gerados para teste. Sendo cada cor uma das 12 regiões, as ruas identificadas pelo número nas arestas:

Obs: Os segmentos têm comprimentos variáveis, mas que não foram representados no gráfico, embora interfiram no menor caminho.

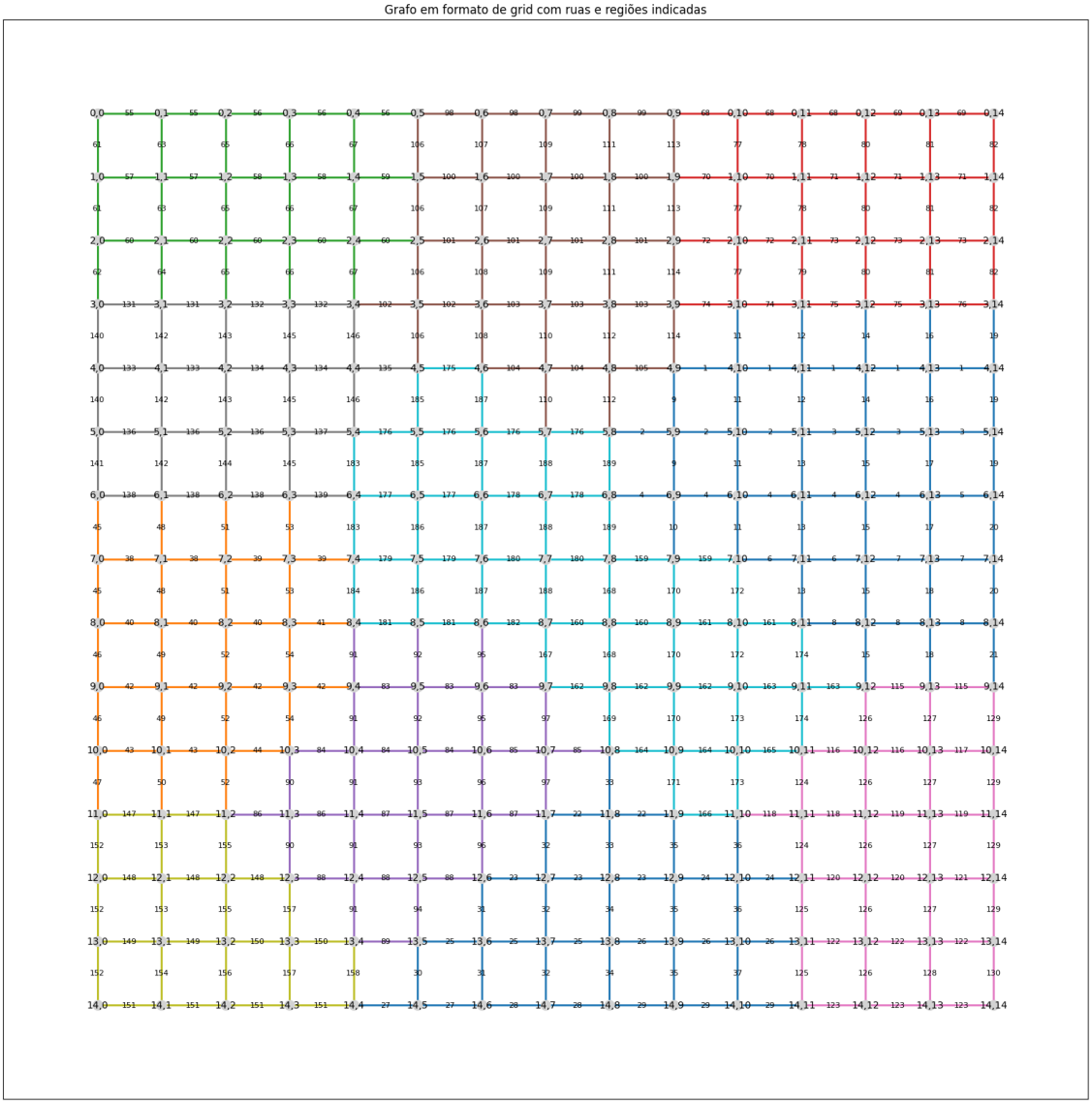

Figura 1: Exemplo de grafo gerado com 12 regiões e ruas definidas

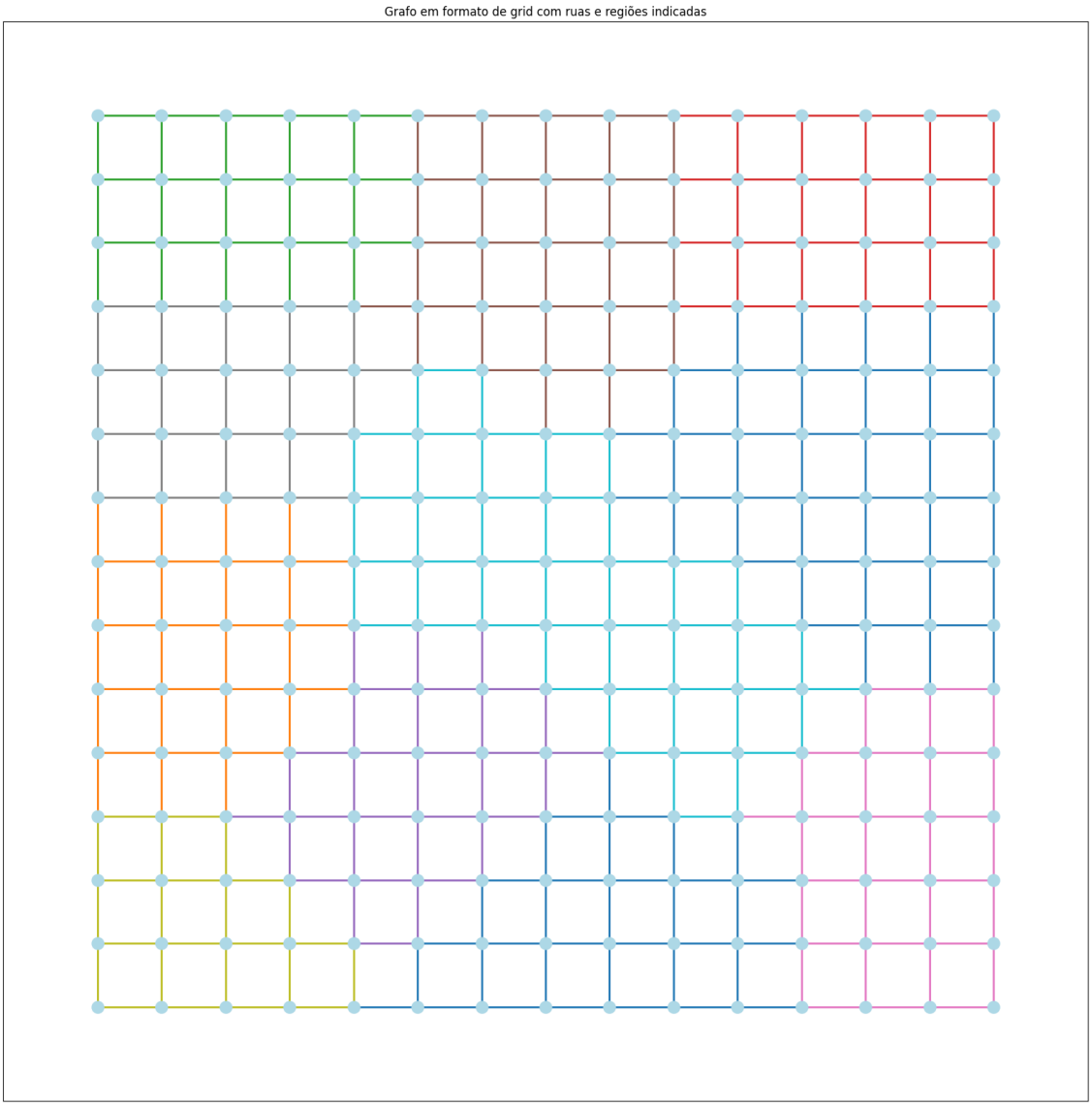

Figura 2: Exemplo de grafo gerado com 12 regiões

Tarefa 1

2.1 Parte 1: A escolha de cada estação de metrô

A escolha de cada estação de metrô deverá buscar minimizar a distância entre ela e o ponto mais longe da sua respectiva região

Dado o grafo não direcionado fortemente conectado que representa a cidade dividida em regiões, onde cada vértice corresponde a um cruzamento na cidade e cada aresta possui um peso indicando a distância, encontrar para cada região o vértice que minimiza a maior distância até qualquer outro vértice da mesma região.

- Dado o planejamento da cidade, recebemos um vetor contendo as arestas e vértices de cada região.
- Como não temos arestas com distâncias negativas, iremos, para cada região, utilizar Dijkstra para calcular as distâncias de todos os vértices dentro dessa região para todos os outros vértices da região. Para cada vértice, selecionamos o vértice cujo caminho tem a maior distância.
- Tendo, para cada vértice seu respectivo de maior distância, encontramos qual desses pares é o que minimiza essa maior distância.

2.1.1 Pseudocódigo

```
// Função cptDijkstraFast
cptDijkstraFast(v0, parent, distance, adjacencyList) //O((V+E)log(V)
    numVertices // tamanho(adjacencyList)
    checked // vetor de booleanos inicializado com false (tamanho numVertices)

    parent // vetor inicializado com -1 (tamanho numVertices)
    distance // vetor inicializado com infinito (tamanho numVertices)

    // Priority queue para gerenciar os vértices
    priority_queue // nova fila de prioridade ordenada por menor valor

    // Define parent do vértice inicial ele mesmo e distancia como 0
    parent[v0.id()] = v0.id()
    distance[v0.id()] = 0
    priority_queue.push((0, v0.id())) // Adiciona na fila

    enquanto priority_queue não está vazia
        v1 = priority_queue.top().segundo
        priority_queue.pop()

        se checked[v1] continuar
        checked[v1] = true

        // Explora as arestas saindo de v1
        para cada tupleData em adjacencyList[v1]
            v2 = tupleData.primeiro
            edge = tupleData.segundo
            cost = edge.distance()
```

```
            // Relaxamento
            se distance[v1] + cost < distance[v2]
                parent[v2] = v1
                distance[v2] = distance[v1] + cost
                pq.push((distance[v2], v2))


// Função findOptimalVertex
findOptimalVertex(regionVertices, adjacencyList) //O(V/r (V+E)logV),
// sendo r = nº de regiões, já que são proporcionais por construção
    numVertices // tamanho(adjacencyList)
    minMaxDist // infinito
    optimalVertex // nulo

    para cada vertexList em regionVertices
        para cada edge em vertexList
            vertex = edge.vertex1()
            parent = vetor inicializado (tamanho numVertices)
            distance = vetor inicializado (tamanho numVertices)

            cptDijkstraFast(vertex, parent, distance, adjacencyList)

            maxDist = -infinito
            para cada innerVertexList em regionVertices
                para cada innerEdge em innerVertexList
                    target = innerEdge.vertex1()
                    maxDist = max(maxDist, distance[target.id()])

            se maxDist < minMaxDist
                minMaxDist = maxDist
                optimalVertex = vertex

    // Define o vértice ótimo encontrado como estação de metro
    se optimalVertex não for nulo
        optimalVertex.setMetroStation(true)

    retornar optimalVertex
```

2.1.2 Análise da complexidade

- Dijkstra por vértice: O((V+E)logV), onde V é o número de vértices e E o número de arestas. Como o grafo é esparso, temos O(VlogV).
- Para n vértices em uma região, a complexidade total é O(n(V+E)logV).
- Fazendo isso para c regiões, temos O(c(n(V+E)logV)) = O(c((V/c)(V+E)logV)) = O(c((V/c)(V)logV)), já que nossas regiões são proporcionais.
- Complexidade total: O($V^2 logV$)

- Complexidade de espaço: O(V+E)

2.1.3 Análise de tempo de testes

Dadas a complexidade já calculada da função findOptimalVertexFast, segue o gráfico, feito em python, que plota a curva da complexidade calculada na teoria e a curva de tempo de execução real da função a fim de verificar se a complexidade calculada está correta:

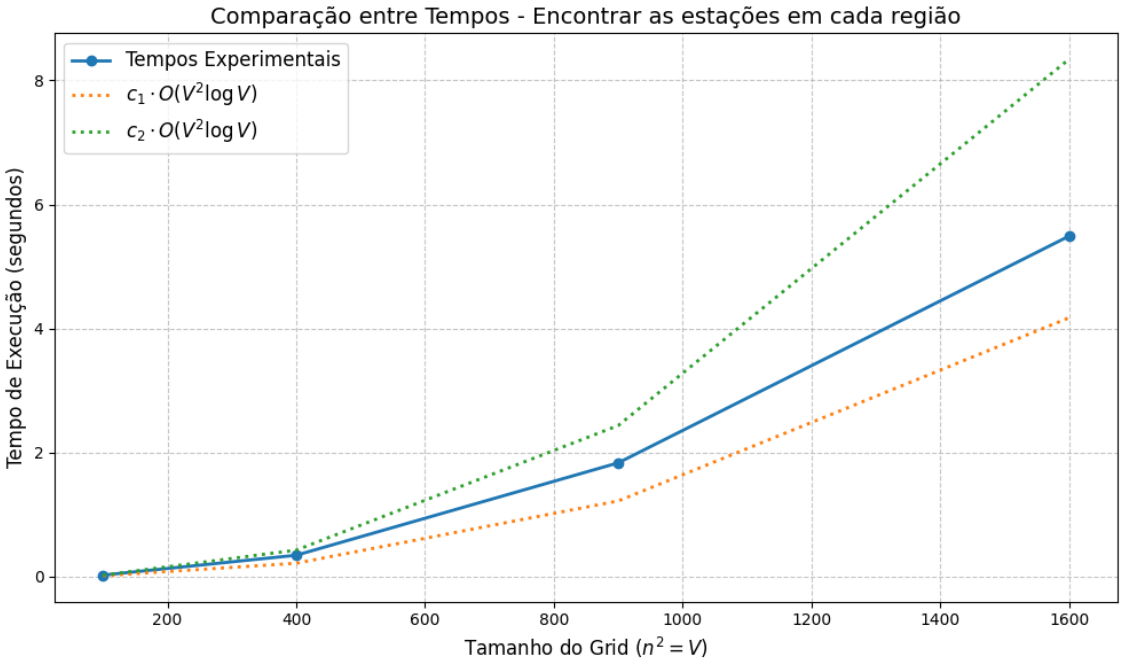

Figura 3: Comparação entre tempos experimentais e teóricos variando o número de vértices

O gráfico apresentado ilustra a comparação entre os tempos de execução experimentais e o crescimento teórico esperado da complexidade $O(V^2 \log V)$, onde $V = n^2$ representa o número total de vértices em uma grid de tamanho $n \times n$.

- A linha azul sólida com marcadores circulares representa os tempos experimentais obtidos em diferentes tamanhos de grid (10×10, 20×20, 30×30, e 40×40). Esses tempos refletem medições reais de execução do algoritmo. - As linha tracejada corresponde à função teórica $O(V^2 \log V)$ com uma constante. A função do tempo é cotada inferior e superiormente por funções múltiplas de $(V^2 logV)$, o que mostra que a função tempo tem também essa complexidade

O tempo de execução experimental segue o mesmo padrão de crescimento da função teórica. Os tempos experimentais estão bastante próximos da previsão teórica.

2.1.4 Exemplo do resultado

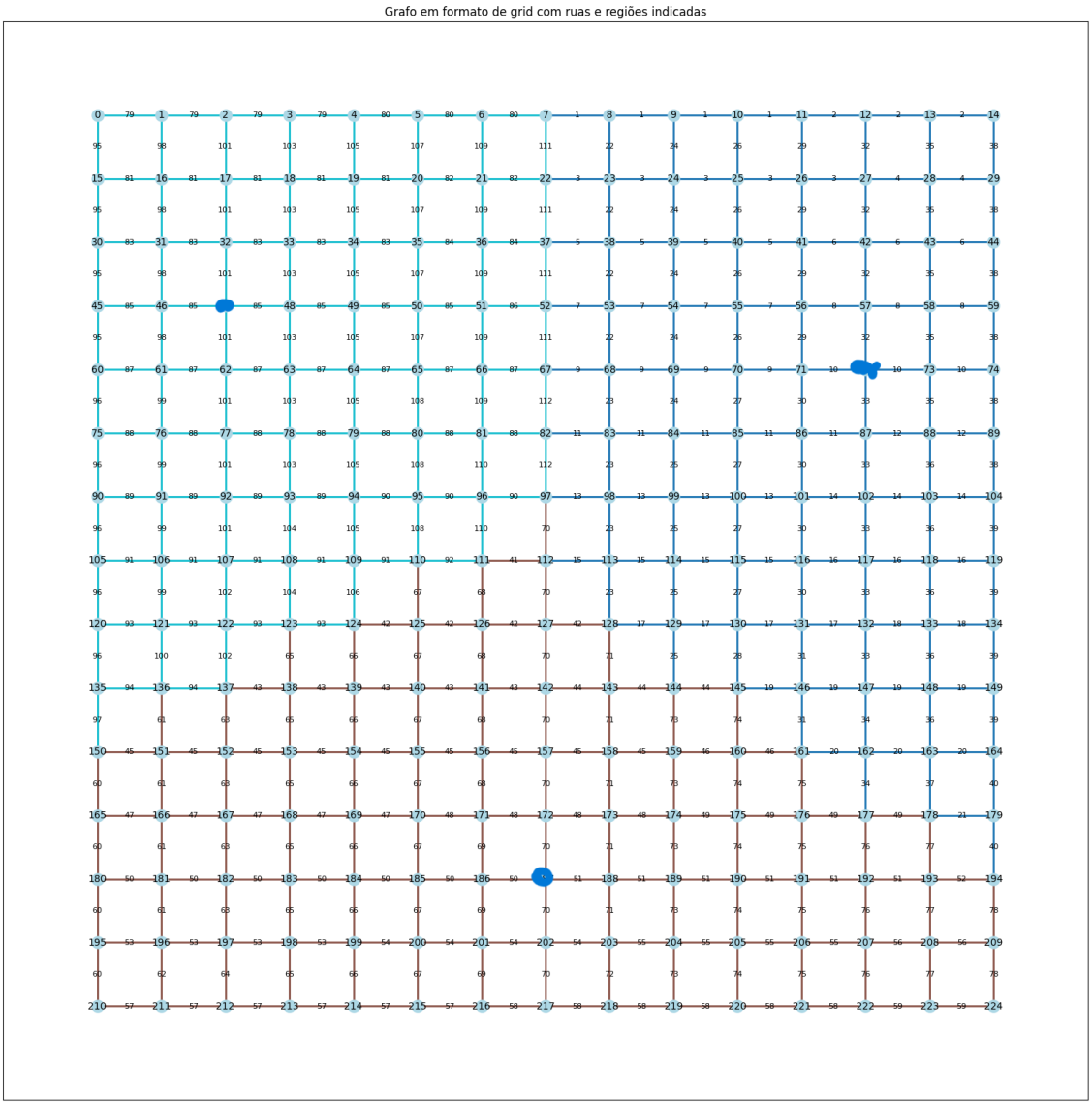

Figura 4: Exemplo de escolha das estações com 3 regiões

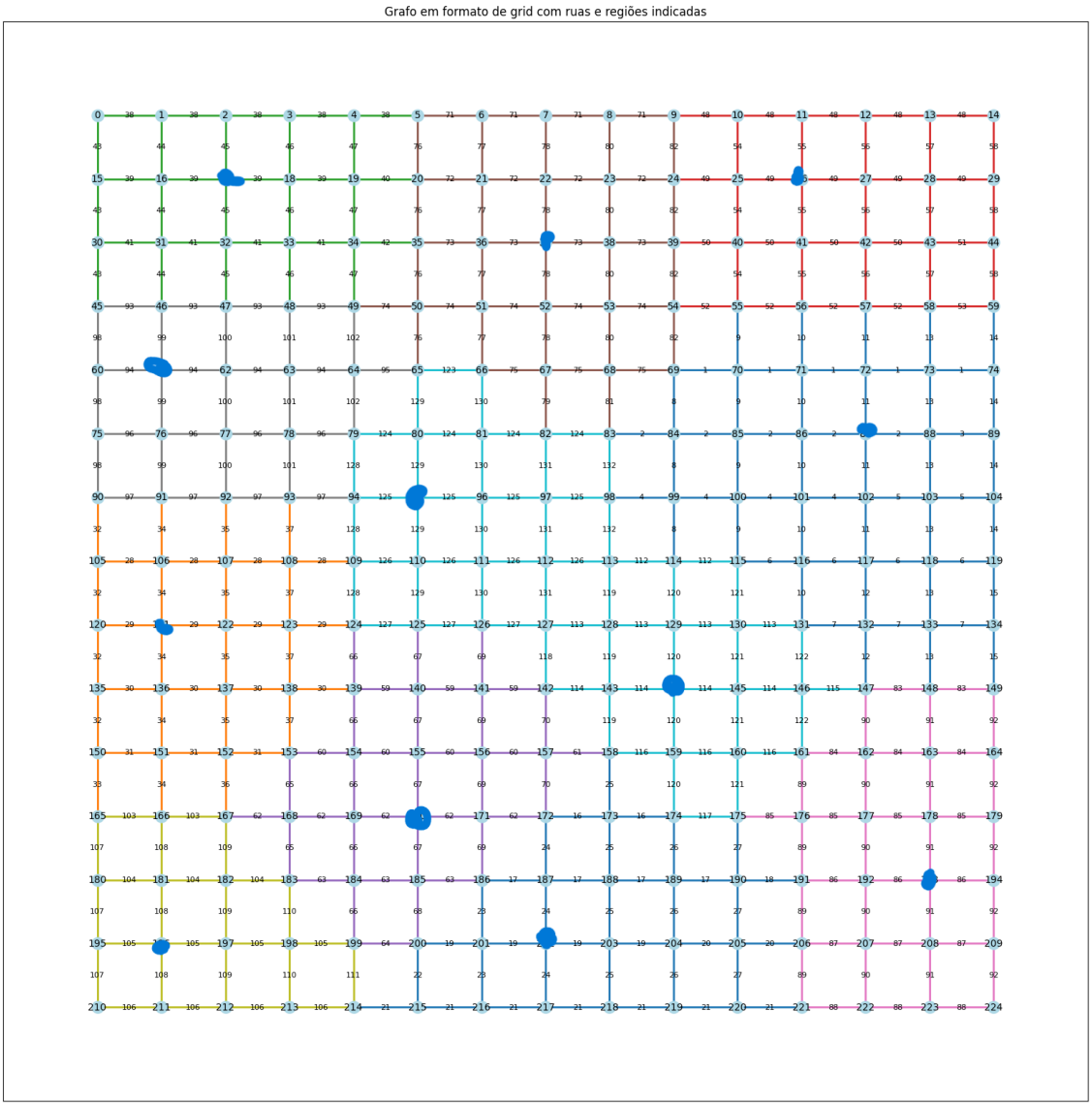

Figura 5: Exemplo de escolha das estações com 12 regiões

2.2 Parte 2: A definição dos segmentos a serem escavados

Projete um algoritmo capaz de definir os segmentos a serem escavados, de forma que o custo para a cidade seja mínimo, no entanto todas as estações definidas sejam conectadas.

O código abaixo cria um sistema otimizado de linhas de metrô conectando uma série de estações (também chamado de terminais), minimizando o custo total de construção. Esse problema é resolvido usando como base o algoritmo da **Árvore de Steiner** que combina Dijkstra e Kruskal para garantir que as conexões entre estações sejam feitas com base nos menores custos de construção.

O grafo do metrô é representado por uma lista de adjacência, onde cada aresta conecta duas estações (vértices) e possui um custo associado (custo de escavação).

Dijkstra:

- Usa-se o algoritmo para calcular o caminho de menor custo entre cada par de estações terminais.
- Cada caminho curto é reconstruído e armazenado para futura referência. Essa etapa assegura que os trajetos utilizados para conectar estações minimizem o custo de construção.

Criação de Arestas Virtuais:

- Para cada par de estações de metro, cria-se uma "aresta virtual"que representa o custo total do menor caminho entre eles, calculado previamente com Dijkstra.
- Essas arestas virtuais formam uma versão reduzida do problema original, onde os custos entre terminais são agregados.

Kruskal:

- O algoritmo é aplicado às arestas virtuais para encontrar uma árvore geradora mínima, ou seja, uma estrutura que conecta todos os terminais com o menor custo total possível, evitando ciclos.
- A solução resultante corresponde à Árvore de Steiner aproximada, pois também considera vértices intermediários no grafo original que ajudam a reduzir os custos.

O sistema retorna não só as arestas da árvore geradora, mas também os caminhos detalhados que compõem essas arestas, ajudando na visualização e execução do projeto de construção. Ou seja, como o projeto exige conectar n estações principais, o algoritmo encontra os melhores caminhos entre elas considerando custos e eficiência.

2.2.1 Pseudocódigo

```
// Função de Dijkstra para calcular o menor caminho entre os vértices
Função Dijkstra(adjacencyList, start):
    // Inicializa o número de vértices no grafo
    n = tamanho de adjacencyList

    // Inicializa as distâncias e os pais
    dist = vetor de tamanho n com valores infinitos (INT_MAX)
    parent = vetor de tamanho n com valores -1

    // Fila de prioridade (heap-mínimo)
    pq = fila de prioridade vazia
```

```
    dist[start] = 0
    pq.push({0, start})

    enquanto pq não estiver vazio:
        u = vértice com a menor distância na fila
        d = distância associada ao vértice u
        pq.pop()

        se d > dist[u]:
            continue  // Ignora se a distância registrada já é maior do que a atual

        // Percorre todos os vizinhos do vértice u
        para cada adjacente adj em adjacencyList[u]:
            v = vértice adjacente
            edge = aresta entre u e v
            weight = peso da aresta (usando edge->distance())

            //Relaxamento das arestas
            se dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                parent[v] = u
                pq.push({dist[v], v})

    Retorne parent


// Função de Kruskal para encontrar a Árvore Geradora Mínima (MST)
Função Kruskal(numVertices, edges):
    mstEdges <- vetor vazio  // Vetor para armazenar as arestas da MST
    // Estrutura de dados Union-Find para evitar ciclos
    uf = instância de UnionFind(numVertices)

    // Ordena as arestas pelo peso (custo de escavação)
    sortedEdges = ordenar(edges, por excavationCost)

    para cada aresta em sortedEdges:
        u = vértice de origem da aresta
        v = vértice de destino da aresta

        // Se os vértices u e v não estão no mesmo conjunto (não formam ciclo)
        se uf.find(u) != uf.find(v):
            mstEdges.push_back(aresta)
            uf.unite(u, v)

    Retorne mstEdges


// Função principal para calcular a Árvore de Steiner com custos agregados
Função ConectMetro(vertices, adjacencyList, terminals, detailedPaths):
    allEdges = vetor vazio
    detailedPaths.clear()
```

```
    new_id_edge = 0

    // Calcula os caminhos mais curtos entre todos os pares de terminais
    allShortestPaths = vetor de tamanho do número de terminais
    para cada terminal t em terminais:
        // Calcula o caminho mais curto de t para todos os outros terminais
        allShortestPaths[t] = Dijkstra(adjacencyList, t.id())

    // Constrói as arestas agregadas e registra os caminhos detalhados
    para cada par de terminais (i, j) em terminais:
        path = reconstruirCaminho(allShortestPaths[i], terminais[j].id())

        pathEdges = vetor vazio
        totalExcavationCost = 0

        para cada aresta entre os vértices u e v no caminho:
            totalExcavationCost = totalExcavationCost + custo de escavação da aresta

            pathEdges.push_back(aresta)  // Adiciona a aresta ao caminho

        // Se o custo de escavação for positivo, cria uma nova aresta agregada
        se totalExcavationCost > 0:
            novaAresta = criarNovaAresta(totalExcavationCost, terminais[i], terminais[j])
            allEdges.push_back(novaAresta)
            detailedPaths.push_back(pathEdges)

    // Aplica o algoritmo de Kruskal para calcular a MST das arestas agregadas
    connectedEdges = Kruskal(vertices.size(), allEdges)

    Retorne connectedEdges


// Função para reconstruir o caminho mais curto a partir dos pais
Função ReconstruirCaminho(parent, destination):
    path = vetor vazio  // Vetor para armazenar o caminho reconstruído
    v = destination

    // Reconstruir o caminho a partir dos pais
    enquanto v != -1:
        path.push_back(v)  // Adiciona o vértice ao caminho
        v = parent[v]  // Vai para o pai de v

    path.reverse()
    Retorne path
```

2.2.2 Análise da complexidade

- **dijkstra**:
  - **Complexidade de tempo:** $O((V + E) \log V)$, onde V é o número de vértices e E é o número de arestas. A fila de prioridade usa operações de inserção e remoção de menor elemento com complexidade $O(\log V)$, executadas para cada vértice e

aresta.

  - **Complexidade de espaço:** $O(V + E)$, para armazenar o vetor de distâncias, os pais dos vértices e a lista de adjacência.

- **kruskal**:

  - **Complexidade de tempo:** $O(E \log E + V \log V)$, onde E é o número de arestas e V é o número de vértices. O termo $O(E \log E)$ refere-se à ordenação das arestas, e $O(V \log V)$ ao uso da estrutura Union-Find para unir e encontrar conjuntos.
  - **Complexidade de espaço:** $O(E + V)$, para armazenar a lista de arestas, os conjuntos disjuntos e os vértices.

- **conect_metro**:

  - **Complexidade de tempo:** $O(T^2 \cdot ((V + E) \log V) + E \log E + V \log V)$, onde T é o número de terminais, V o número de vértices e E o número de arestas. O termo $O(T^2 \cdot ((V + E) \log V))$ refere-se ao cálculo de caminhos mais curtos entre todos os pares de terminais, e os outros termos ao Kruskal aplicado nas arestas agregadas.
  - **Complexidade de espaço:** $O(T^2 + V + E)$, para armazenar os caminhos mais curtos entre terminais, os vértices e as arestas.

- **reconstructPath**:

  - **Complexidade de tempo:** $O(P)$, onde P é o número de vértices no caminho reconstruído. A reconstrução percorre os vértices do destino até a origem.
  - **Complexidade de espaço:** $O(P)$, para armazenar o vetor de caminho reconstruído.

2.2.3 Análise de tempo de testes

Dadas as complexidades já calculadas das funções acima, que descobrem o melhor caminho para conectar as estações de metrô, segue o gráfico, feito em python, que plota a curva da complexidade calculada na teoria e a curva de tempo de execução real da função a fim de verificar se a complexidade calculada está correta:

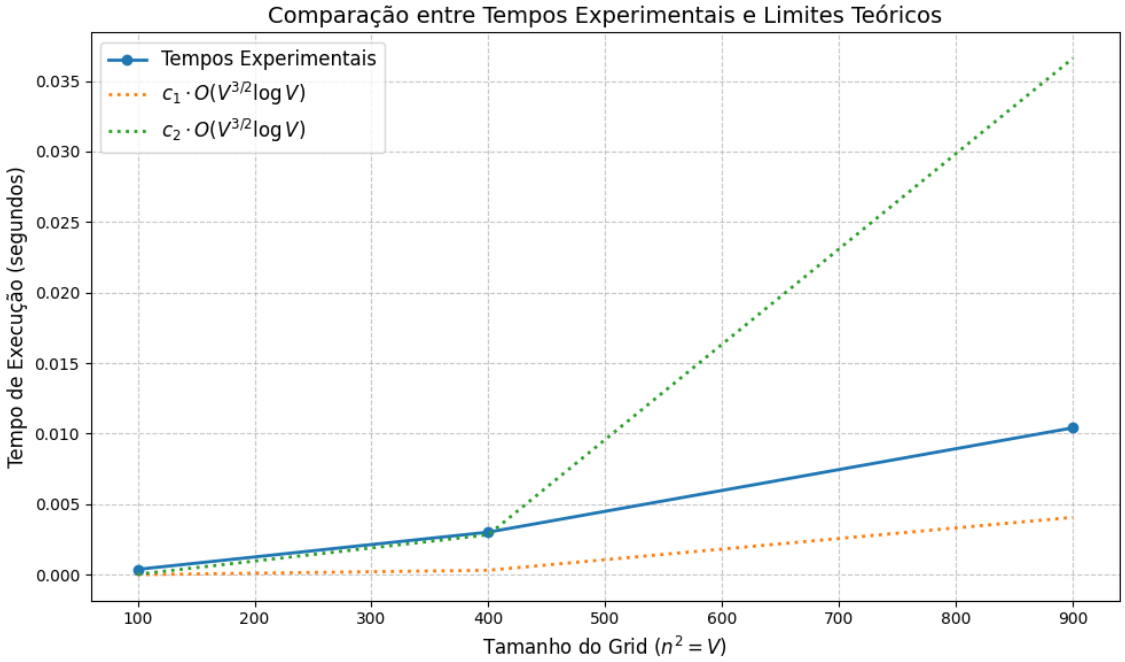

Figura 6: Comparação entre tempos experimentais e teóricos variando o número de vértices

O gráfico apresentado ilustra a comparação entre os tempos de execução experimentais e o crescimento teórico esperado da complexidade $O(V^{3/2} \log V)$, onde $V = n^2$ representa o número total de vértices em uma grid de tamanho $n \times n$.

- A linha azul sólida com marcadores circulares representa os tempos experimentais obtidos em diferentes tamanhos de grid (10×10, 20×20, 30×30, e 40×40). Esses tempos refletem medições reais de execução do algoritmo. A função do tempo é cotada inferior e superiormente por funções múltiplas, confirmando a complexidade teórica.

2.2.4 Exemplo do resultado

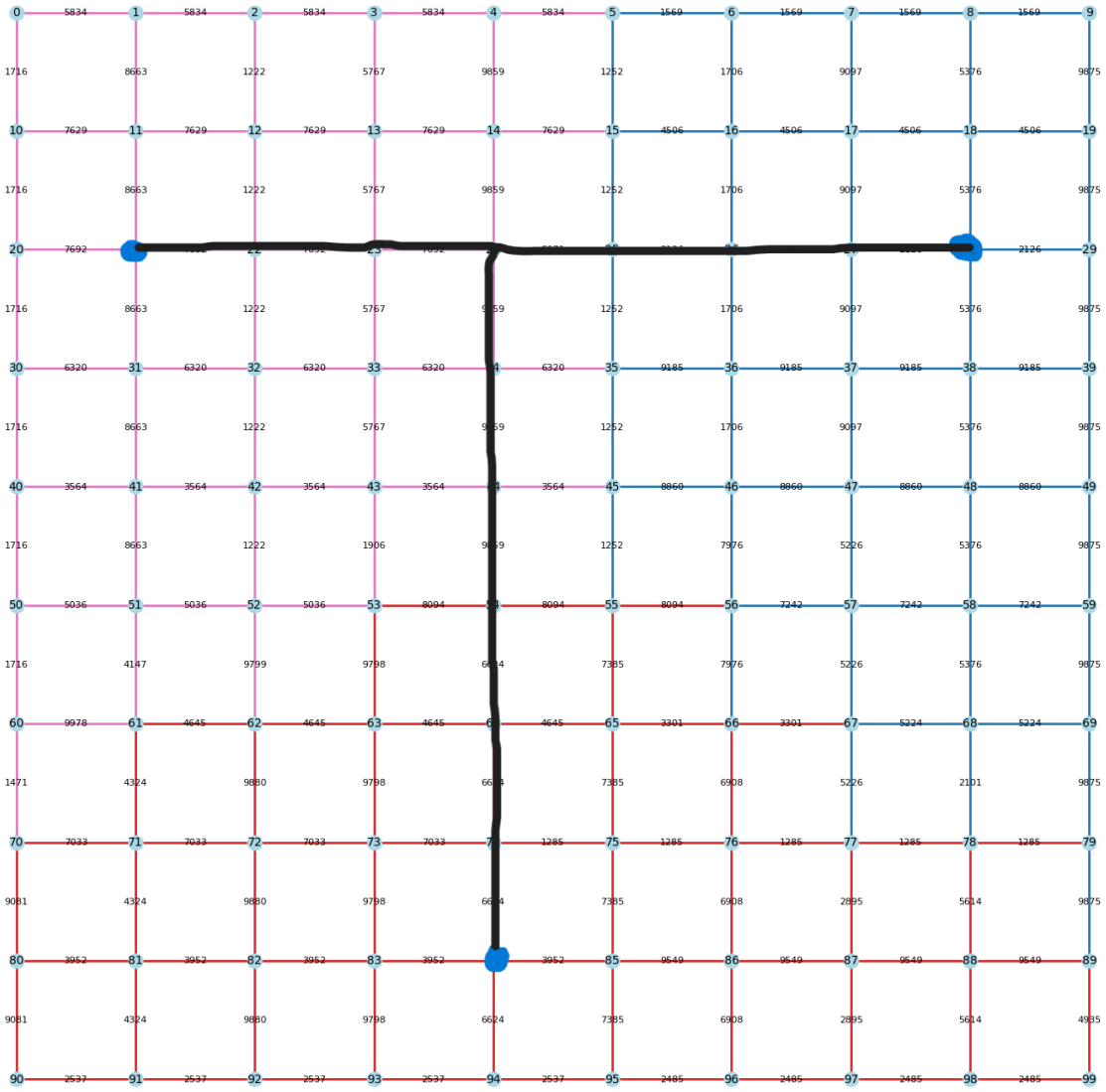

Figura 7: Exemplo de linha de metro com 3 estações

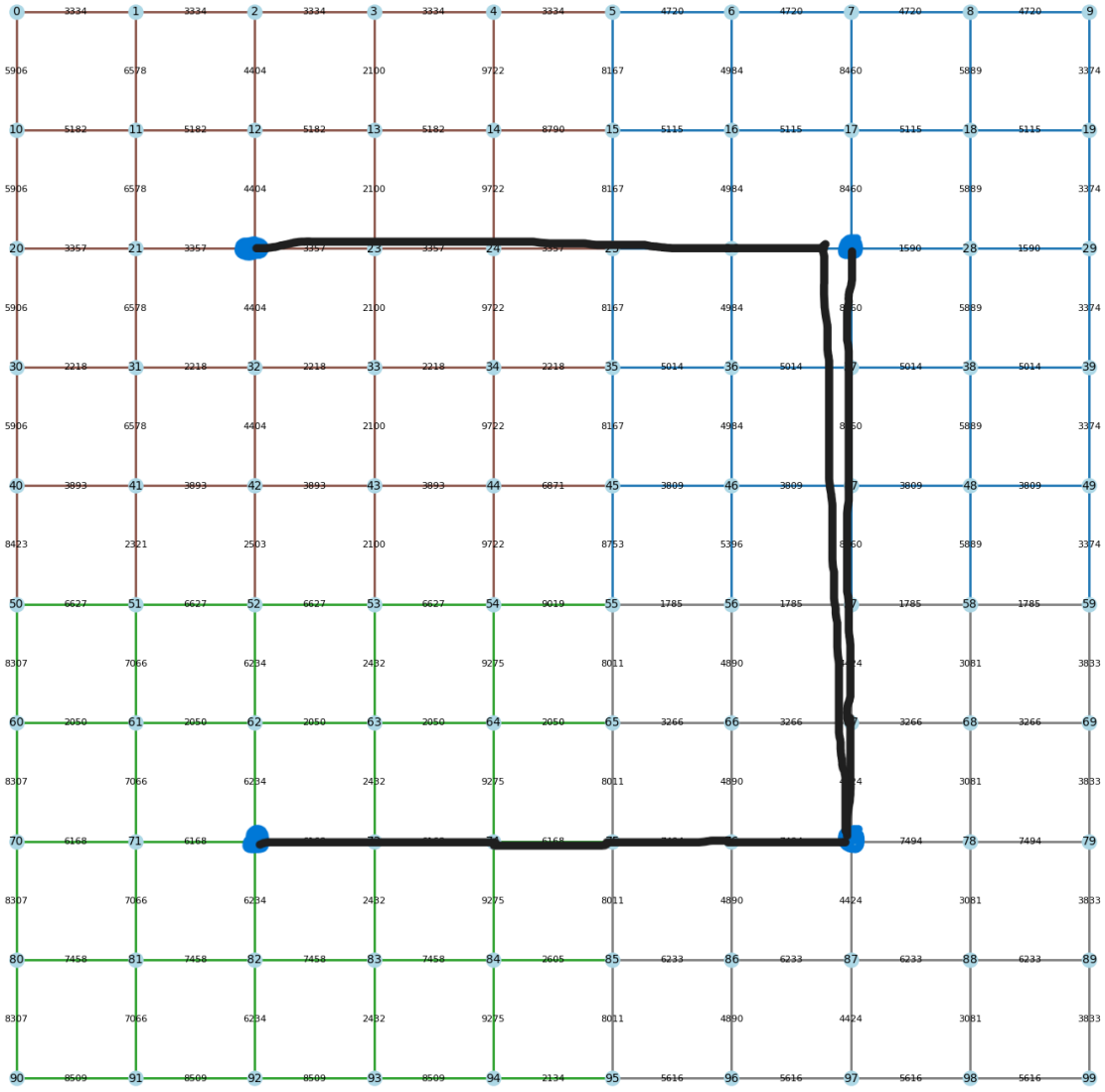

Figura 8: Exemplo de linha de metro com 4 estações

Tarefa 3

3.1 Parte 1: Gerar um grafo direcionado para as rotas de táxis

Primeiramente geramos um grafo não direcionado para construir as linhs de metrô pois essas não levam em conta as direções das ruas, mas para definir as rotas dos táxis devemos ter um grafo direcionado para dizer as direções válidas para aquela aresta. Assim, foi definida uma função que recebe o grafo não direcionado inicialmente criado e percorre esse grafo com bfs e para cada aresta visitada assimila uma direção seguindo a seguinte lógica:

- se o vértice tem grau de entrada 0 então adiciona uma aresta apontando para o vértice corrente, atalizamos a lista de adjacência do vértice corrente;
- se o seu grau de saída é 0 então adiciona uma aresta apontando para o vértice na outra extremidade, atualizamos a lista de adjacência do vértice extremo;
- e se o grafo já tiver grau de entrada e saída maiores que um e restaram arestas, estas terão direções aleatórias, podendo ser de saída ou entrada.

Além disso para implementar as ruas de mão dupla cada associação de direção pode criar uma outra aresta na direção oposta.

Como o grafo original tem todos os seus vértices com pelo menos outros dois vértices adjacentes distintos, ou seja não há vértices isolados, essa lógica de assimilação de direção garante que teremos caminho entre todos os vértices., pois evita a definição de sorvedouros ou fontes forçando todo vértice a ter pelo menos uma aresta de saída e entrada.

3.1.1 Pseudocódigo

```
// Percorre por bfs adicionando direcao ao grafo para arestas percorridas
directingBfs(start, listAdj, directedAdj, degreeOut, degreeIn)
    // Armazenar acessos a vertex e edges
    list visited                    //} custo de espaço V
    list processedEdges             //} custo de espaço E

    // Fila para a BFS
    queue fila
    fila.push(start)
    visited.insert(start)

    enquanto fila não vazia                 // custo O(V)
        // Primeiro vertex na fila          // percorre todos os vértices e arestas
        current = fila.front()              // adicionando as direções, O(1)
        fila.pop()

        // acessa os adjacentes do vértice atual
        para cada aresta em current             // O(E)
            só processar arestas não visitadas
            processedEdges.insert(aresta)                   //O(1)
            nextVertex = otherVertex(aresta, current)       //O(1)
            se grau entrada 0
                adiciona uma aresta em direção ao vértice "current"
                // é uma lista encadeada dos vértices adjacentes, então a inserção é O(1)
                directedAdj[current].insert(nextVertex)
```

```
                e atualiza os graus de entrada e saída dos vértices         //O(1)

                    Adiciona uma  aresta na direção oposta aleatoriamente
                    // essas são as ruas de mão dupla
                    if aleatorio == True
                        cria nova aresta com direção oposta, apontando para o nextVertex
                        Atualiza os graus de entrada e saída

            se grau de saída 0                      // todas as operações são O(1)
                adiciona uma aresta em direção ao vértice "nextVertex"
                directedAdj[nextVertex].insert(current)
                Atualiza os graus de vértice

                Adiciona uma  aresta na direção oposta aleatoriamente
                if aleatorio == True
                    cria nova aresta com direção oposta, apontando para o current
                    Atualiza os graus de entrada e saída

            se já tem grau de entrada e saída > 1 adiciona uma aresta com direção aleatória
            aresta apontando para nextVertex ou current


            Se o próximo vértice ainda não foi visitado, adiciona na fila

// Rebebe um grafo não direcionado e percorre com directingBfs
convertToDirected(adj)
    // Inicializa estruturas auxiliares
    nova lista de adjacência directedAdj        //O(V+E) de espaço
    list visited;                               //O(V)
    list processedEdges;                        // O(E)

    // Inicializa os vetores de graus com zero
    degreeOut[] = 0                             //O(V)
    degreeIn[] = 0                              //O(V)

    Escolhe um vértice inicial start
    // percorre o grafo adicionando as aresta equivalentes mas com direções
    directingBfs(start, listAdj, directedAdj, degreeOut, degreeIn)      //O(V+E)

    return directedAdj
```

3.1.2 Análise da complexidade

O algoritmo de BFS tem complexidade de $O(V + E)$, assim como o algoritmo original percorremos os vértices e adicionamos os vértices adjacentes que ainda não foram processados, para cada vértice avalia cada aresta e pode apontar essa aresta para o vértice corrente ou para o vértice na outra extremidade da aresta dependendo do grau de entrada e saída do vértice que está sendo iterado, até aqui acessamos as listas de adjacências e adicionamos qual o vértice destino, além de calcular se adicionamos uma aresta nova tabém é uma operação constante.

A função final "convertToDirect"inicia as estruturas auxiliares para acompanhar as iterações, essas estruturas tem custo de espaço $O(V+E)$, as listas de vértice vistados e arestas

processadas além dos vetores de graus de saída e entrada, e a complexidade de processamento é a mesma pois o algoritmo de BFS tem o custo de $O(V + E)$ já que percorre todo vértice e toda aresta fazendo operações constantes.

3.2 Parte 2: Gerar o algoritmo que faça a rota do táxi

A função implementa o algoritmo de Dijkstra para calcular o menor caminho em um grafo, adaptado para um cenário de transporte por táxi. O objetivo é determinar o caminho mais rápido entre dois pontos (vértices) de um mapa, levando em consideração fatores como tráfego, distância e custo da corrida.

O grafo é representado como uma lista de adjacência, onde cada aresta contém informações sobre a distância e uma taxa de tráfego, que ajusta a velocidade máxima permitida. A função começa no ponto inicial e explora as conexões disponíveis, calculando o tempo necessário para percorrer cada estrada com base na velocidade ajustada e a distância. Além disso, o custo da corrida é atualizado, considerando a distância percorrida e a tarifa do táxi por metro.

Durante a execução, a função mantém o controle dos menores tempos e custos conhecidos para cada ponto do mapa, priorizando sempre os caminhos mais rápidos. Ela utiliza uma fila de prioridade para garantir que os vértices sejam processados na ordem de menor tempo acumulado.

Ao final, a função retorna:

- O tempo total da viagem (em minutos).
- O custo total estimado da corrida de táxi.
- A sequência de estradas percorridas (identificadas por seus IDs).

Se não houver um caminho viável entre os pontos especificados, a função informa que o trajeto não é possível.

3.2.1 Pseudocódigo

```
// Função de Dijkstra para calcular o menor caminho entre os vértices
Função Dijkstra(adjacencyList, start):
    // Inicializa o número de vértices no grafo
    n = tamanho de adjacencyList

    // Inicializa as distâncias e os pais
    dist = vetor de tamanho n com valores infinitos (INT_MAX)
    parent = vetor de tamanho n com valores -1

    // Fila de prioridade (heap-mínimo)
    pq = fila de prioridade vazia
    dist[start] = 0
    pq.push({0, start})

    enquanto pq não estiver vazio:
        u = vértice com a menor distância na fila
        d = distância associada ao vértice u
        pq.pop()

        se d > dist[u]:
            continue  // Ignora se a distância registrada já é maior do que a atual
```

```
        // Percorre todos os vizinhos do vértice u
        para cada adjacente adj em adjacencyList[u]:
            v = vértice adjacente
            edge = aresta entre u e v
            weight = peso da aresta (usando edge->distance())

            //Relaxamento das arestas
            se dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                parent[v] = u
                pq.push({dist[v], v})

    Retorne parent


// Função de Kruskal para encontrar a Árvore Geradora Mínima (MST)
Função Kruskal(numVertices, edges):
    mstEdges = vetor vazio  // Vetor para armazenar as arestas da MST
    // Estrutura de dados Union-Find para evitar ciclos
    uf = instância de UnionFind(numVertices)

    // Ordena as arestas pelo peso (custo de escavação)
    sortedEdges = ordenar(edges, por excavationCost)

    para cada aresta em sortedEdges:
        u = vértice de origem da aresta
        v = vértice de destino da aresta

        // Se os vértices u e v não estão no mesmo conjunto (não formam ciclo)
        se uf.find(u) != uf.find(v):
            mstEdges.push_back(aresta)
            uf.unite(u, v)

    Retorne mstEdges


// Função principal para calcular a Árvore de Steiner com custos agregados
Função ConectMetro(vertices, adjacencyList, terminals, detailedPaths):
    allEdges = vetor vazio
    detailedPaths.clear()

    new_id_edge = 0

    // Calcula os caminhos mais curtos entre todos os pares de terminais
    allShortestPaths = vetor de tamanho do número de terminais
    para cada terminal t em terminals:
        // Calcula o caminho mais curto de t para todos os outros terminais
        allShortestPaths[t] = Dijkstra(adjacencyList, t.id())

    // Constrói as arestas agregadas e registra os caminhos detalhados
    para cada par de terminais (i, j) em terminals:
```

```
            path = reconstruirCaminho(allShortestPaths[i], terminals[j].id())

            pathEdges = vetor vazio
            totalExcavationCost = 0

            para cada aresta entre os vértices u e v no caminho:
                totalExcavationCost = totalExcavationCost + custo de escavação da aresta

                pathEdges.push_back(aresta)  // Adiciona a aresta ao caminho

            // Se o custo de escavação for positivo, cria uma nova aresta agregada
            se totalExcavationCost > 0:
                novaAresta = criarNovaAresta(totalExcavationCost, terminais[i], terminais[j])
                allEdges.push_back(novaAresta)
                detailedPaths.push_back(pathEdges)

    // Aplica o algoritmo de Kruskal para calcular a MST das arestas agregadas
    connectedEdges = Kruskal(vertices.size(), allEdges)

    Retorne connectedEdges


// Função para reconstruir o caminho mais curto a partir dos pais
Função ReconstruirCaminho(parent, destination):
    path = vetor vazio  // Vetor para armazenar o caminho reconstruído
    v = destination

    // Reconstruir o caminho a partir dos pais
    enquanto v != -1:
        path.push_back(v)  // Adiciona o vértice ao caminho
        v = parent[v]  // Vai para o pai de v

    path.reverse()
    Retorne path
```

3.2.2 Análise da Complexidade

- **Inicializações:** $O(n)$ para inicializar os vetores.
- **Iteração sobre todos os vértices:** O algoritmo principal visita cada vértice no pior caso, utilizando a fila de prioridade. Isso resulta em $O((V + E) \log V)$, onde:
  - V é o número de vértices.
  - E é o número de arestas.
  - O fator $\log V$ vem das operações de inserção e remoção da fila de prioridade.
- **Relaxamento de arestas:** Cada aresta é processada no máximo uma vez. A análise de uma aresta requer cálculo de tempo e custo, ambos $O(1)$.

Complexidade total de tempo: $O((V + E) \log V) = O(V \log V)$

- **Estruturas principais:** Vetores `minTime`, `totalCost`, `parent`, e `edgeUsed` requerem $O(n)$ de espaço.

- A fila de prioridade pode armazenar no máximo $O(n)$ elementos.

Complexidade total de espaço: $O(V + E)$, onde o espaço adicional para a fila é proporcional ao número de vértices.

3.3 Parte 3: Encontrar o caminho do metrô mais próximo da origem até o mais próximo do destino

Utilizando a MST gerada na parte 2 da tarefa um, temos a linha de metrô e as suas estações. Aqui fazemos uma função que encontra a estação de metro da região da saída do usuário e a estação da região do destino. Retorna os vértices de caminho entre eles, as estações intermediárias nesse caminho e a distância entre elas, na linha de metro.

Porque assim, conseguimos ver se compensa pegar outro trnaporte, saindo em qualquer estação intermediária entre a da região de origem e de destino e temos a distância até cada uma delas.

- Localizar as estações de metrô: Busca na lista de adjacência (mstadj - MST da linha de metro) pelos vértices associados aos CEPs e verifica se eles são estações de metrô.
- DFS: Realiza uma busca em profundidade (DFS) para encontrar o caminho entre as duas estações.
  Armazena o mapeamento dos pais para reconstruir o caminho posteriormente.
- Reconstrução do caminho: Usa o mapa de pais gerado pela DFS para reconstruir o caminho completo, do destino até a origem.
- Filtrar estações de metrô: Seleciona apenas os vértices no caminho que são estações de metrô.
- Calcular custos entre estações: Para cada par consecutivo de estações, percorre o subcaminho correspondente e calcula o custo acumulado.
- Retornar os resultados: Retorna o caminho completo, os custos segmentados e as estações.

3.3.1 Pseudocódigo

```
function findPathBetweenStation(mstadj, region1CEP, region2CEP):
    // Inicializa variáveis
    path = []               // Caminho completo
    segmentDistances = []   // Custos entre estações de metrô
    stations = []           // Estações de metrô no caminho

    // Localizar as estações de metrô associadas aos CEPs
    station1 = NULL
    station2 = NULL
    for each vertex in mstadj: //O(V), V = vértices da MST da linha de metro
        for each (neighbor, edge) in mstadj[vertex]:
            if edge.zipCode == region1CEP and edge.vertex1.isMetroStation:
                station1 = edge.vertex1
            if edge.zipCode == region2CEP and edge.vertex2.isMetroStation:
                station2 = edge.vertex2
        if station1 and station2:
            break

    if station1 is NULL or station2 is NULL:
        throw "Estações não encontradas"
```

```
// Realizar DFS para encontrar o caminho
parent = map()  // Mapeia cada vértice ao seu pai
visited = set()

function dfs(current):
    mark current as visited
    if current == station2.id:
        return True
    for each (neighbor, edge) in mstadj[current]:
        if neighbor not in visited:
            parent[neighbor] = current
            if dfs(neighbor):
                return True
    return False

parent[station1.id] = -1
if not dfs(station1.id):
    throw "Caminho não encontrado"

// Reconstruir o caminho a partir do mapa de pais
current = station2.id
while current != -1:
    path.append(current)
    current = parent[current]
reverse(path)

// Filtrar estações de metrô no caminho
for vertex in path:
    if vertex.isMetroStation:
        stations.append(vertex)

// Calcular custos entre estações
for i from 1 to stations.size - 1:
    startStation = stations[i - 1]
    endStation = stations[i]
    segmentCost = 0

    for j from 1 to path.size - 1:
        v1 = path[j - 1]
        v2 = path[j]
        if v1 == startStation or segmentCost > 0:
            for each (neighbor, edge) in mstadj[v1]:
                if neighbor == v2:
                    segmentCost += edge.distance
                    break
        if v1 == endStation:
            break

    segmentDistances.append(segmentCost)

// Retornar o resultado
```

```
    return (path, segmentDistances, stations)
```

3.3.2 Análise da Complexidade

Sendo V e E, o número de vértices e arestas da MST que representa a linha de metro.

Busca das estaçõoes:(O+E)
DFS: O(V+E)
Resonstrução do caminho: O(V)
Filtrar estações: O(V+E)
Calculo dos custos: O(E.V) = O(V^2)

3.4 Parte 4: Gerar rota mínima para deslocamento não-motorizado

Se os meios alternativos não satisfazerem o orçamento então a locomoção deve se dar por meio não motorizado que não tem custo para o usuário, nesse caso não devemos considerar taxa trânsito nem direção das arestas e o caminho que minimiza o tempo será também o de menor distância. Para esse problema foi implementado o algortimo de Dijkstra que retorna a distancia que deve ser percorrida e o tempo que leva percorrendo esse percurso.

3.4.1 Pseudocódigo

```
Função dijkstraFoot(início, destino, listaAdjacência)
    numVertices = tamanho(listaAdjacência)
    checked[vértice] = falso para cada vértice
    parent[vértice] = (-1, nulo) para cada vértice
    distance[vértice] = infinito para cada vértice

    Crie uma fila de prioridade (pq) para gerenciar vértices a serem processados

    parent[início] = (início, nulo)
    distance[início] = 0
    Adicione (0, início) na fila pq

    Enquanto a fila pq não estiver vazia:
        Remova o vértice com a menor distância (v1)
        Se v1 já foi verificado, continue para o próximo
        Marque v1 como verificado

        Para cada aresta conectada a v1:
            Obtenha v2 (vértice vizinho) e a distância da aresta (custo)
            Se a nova distância para v2 for menor que a distância atual:
                Atualize o parent[v2] para (v1, aresta atual)
                Atualize distance[v2]
                Adicione (nova distância, v2) à fila pq

    Calcule o tempo total de percurso:
        tempo = distância do destino / 83.33 (velocidade em metros/minuto)

    Reconstrução do caminho:
        Inicie uma pilha para armazenar as arestas
        v = destino
        Enquanto v não for o início:
            Se v for -1, retorne "Caminho não encontrado"
            Adicione a aresta de parent[v] à pilha
            Atualize v para o vértice anterior em parent

    Converta a pilha para uma lista de arestas
    Retorne a lista de arestas e o tempo total
```

3.4.2 Análise da Complexidade

- **Inicialização:**
  A inicialização das estruturas auxiliares (`checked`, `parent`, `distance`) leva $O(V)$, onde V é o número de vértices.

- **Inserção na Fila de Prioridade:**
  Inserir o vértice inicial na fila de prioridade custa $O(\log V)$.
- **Loop Principal (while):**
  - O loop processa cada vértice uma vez, resultando em $O(V \log V)$ devido à manipulação da fila de prioridade.
  - Para cada vértice, percorremos suas arestas adjacentes, resultando em $O(E \log V)$, onde E é o número de arestas.
- **Reconstrução do Caminho:**
  A reconstrução do caminho percorre no máximo $O(V)$ vértices.

Complexidade Total de Tempo:

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

Complexidade de Espaço:

- Vetores auxiliares (`checked`, `parent`, `distance`) ocupam $O(V)$.
- A fila de prioridade armazena até $O(V)$ elementos.
- A pilha de reconstrução do caminho usa $O(V)$.

Complexidade Total de Espaço:

$$O(V + E)$$

3.5 Parte 5: Verificar caminho de táxi e a pé dado o orçamento e o tempo

A função foi feita para verificar se a viagem de táxi pelo menor caminho seria por um preço menor ou igual ao valor disponível, ou seja, o máximo que o consumidor está disposto a pagar pela viagem até o seu destino. Além disso, se o percurso de táxi for mais caro do que o consumidor pode pagar, analisamos se a menor rota que pode ser feita a pé até o destino gastaria menos tempo que a viagem de carro porque isso quer dizer que não está muito distante, então seria a opção com menor gasto de tempo e custo zero. Caso uma destas opções seja válida como a melhor partindo do ponto em que está até o endereço destino, retorna as arestas que compõem o caminho indicando o meio de transporte em cada uma.

3.5.1 Pseudocódigo

```
Algoritmo findBestPath(início, destino, listaAdjacente,
listaAdjacenteDirecionada, orçamento)
    arestaInicial = início.arestas
    vérticeInicial = início.vértice
    vérticeDestino = destino.vértice

    [custoCarro, tempoCarro, caminhoCarro] = dijkstraTaxi(listaAdjacenteDirecionada,
        vérticeInicial, vérticeDestino)
    Se custoCarro <= orçamento então
        Exibir "Recomendado ir de carro com custo e tempo"
        Retornar caminhoCarro

    [caminhoPe, tempoPe] = dijkstraFoot(vérticeInicial, vérticeDestino, listaAdjacente)
    Se caminhoPe não estiver vazio E tempoPe < tempoCarro então
        Exibir "Recomendado ir a pé com tempo"
        Retornar caminhoPe

Caso contrário:
    Exibir "Nenhum caminho a pé é mais rápido e o custo de carro excede o orçamento"
    Retornar caminho vazio
```

3.5.2 Complexidade de Tempo

- **Dijkstra para o caminho de carro (`dijkstraTaxi`):**
  Como já explicado anteriormente, $O((V + E) \log V)$.
- **Dijkstra para o caminho a pé (`dijkstraFoot`):**
  Como já explicado anteriormente, $O((V + E) \log V)$.
- **Comparação de custos e tempos:**
  As comparações realizadas para decidir qual caminho usar são operações constantes $O(1)$.

Complexidade Total de Tempo:
Como os dois algoritmos de Dijkstra são executados independentemente, a complexidade total é:

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

3.5.3 Complexidade de Espaço

- **Espaço para armazenar listas de adjacência:**
  A lista de adjacência ocupa $O(V + E)$, já que cada vértice e cada aresta são representados uma vez.
- **Espaço utilizado pelo algoritmo de Dijkstra:**
  A fila de prioridade armazena no máximo V vértices, então ocupa $O(V)$. As estruturas auxiliares para custos, pais e distâncias também ocupam $O(V)$.
- **Espaço para o caminho retornado:**
  No pior caso, pode haver E arestas no caminho, resultando em $O(E)$.

Complexidade Total de Espaço:

$$O(V + E)$$

3.6 Parte 6: Calcula a rota final com os meios de transporte usados

3.6.1 Pseudocódigo

```
Função fastestRoute(começo, destino, arestaInicial, arestaFinal, hora,
orçamento, listaAdjacência, listaAdjacênciaDirecional, listaAdjMetro)
    Preço do bilhete do metrô = 10
    Duração total = 0
    Rota = []

    (caminhoCompleto, custo, tempo, locomoção) = findBestPath(começo,
    destino, listaAdjacência, listaAdjacênciaDirecional, orçamento)

    Se (caminhoCompleto não estiver vazio)
        Orçamento = orçamento - custo
        Adicionar caminho e locomoção na rota
        Retornar (rota, tempo, orçamento)

    Se (mesma região ou orçamento < bilhete de metrô)
        (caminhoAPé, tempoAPé) = dijkstraFoot(começo, destino, listaAdjacência)
        Adicionar caminho a pé na rota
        Retornar (rota, tempoAPé, orçamento)

    Orçamento = orçamento - bilhete de metrô
    (caminhoMetrô, distâncias, estações) = findPathBetweenStation(listaAdjMetro,
    arestaInicial, arestaFinal)
    RegiãoMetrô = primeira estação

    (custoTáxi, tempoTáxi, caminhoTáxi) = dijkstraTaxi(listaAdjacênciaDirecional,
    começo, RegiãoMetrô)
    Se (orçamento - custoTáxi >= 0)
        Adicionar caminho de táxi na rota
    Caso contrário
        (caminhoAPé, tempoAPé) = dijkstraFoot(começo, RegiãoMetrô, listaAdjacência)
        Adicionar caminho a pé na rota

    Espera = (hora + duração) % 20
```

```
Duração = duração + espera

Enquanto (existirem estações a visitar)
    DistânciaEntre = próxima distância
    DuraçãoMetrô = DistânciaEntre / velocidade do metrô
    Duração += DuraçãoMetrô
    Atualizar estação atual e rota com caminho de metrô

    (caminho, custo, tempo, locomoção) = findBestPath(estaçãoAtual,
    destino, listaAdjacência, listaAdjacênciaDirecional, orçamento)
    Se (caminho não estiver vazio)
        Adicionar caminho na rota
        Retornar (rota, tempo + duração, orçamento - custo)

(caminhoFinal, custoFinal, tempoFinal, locomoçãoFinal) = findBestPath(estaçãoAtual,
destino, listaAdjacência, listaAdjacênciaDirecional, orçamento)
Se (caminhoFinal não estiver vazio)
    Adicionar caminho na rota
    Retornar (rota, tempoFinal + duração, orçamento - custoFinal)

(caminhoAPé, tempoAPé) = dijkstraFoot(estaçãoAtual, destino, listaAdjacência)
Adicionar caminho a pé na rota
Retornar (rota, tempoAPé + duração, orçamento)
```

3.6.2 Análise da Complexidade

Considerando as análises das funções usadas calculadas anteriormente e também que o número de regiões na nossa modelagem é $V^2/2 = \mathrm{S}$
Complexidade Total: $= O(V^2 + SVlogV) = O(V^2logV)$

3.6.3 Análise de tempo de testes

Dadas as complexidades já calculadas das funções acima, que descobrem o melhor caminho para sair de um local e ir até outro no grafo usando os meios de transporte de modo a otimizar custo em dinheiro e em tempo, segue o gráfico, feito em python, que plota a curva da complexidade calculada na teoria e a curva de tempo de execução real da função a fim de verificar se a complexidade calculada está correta:

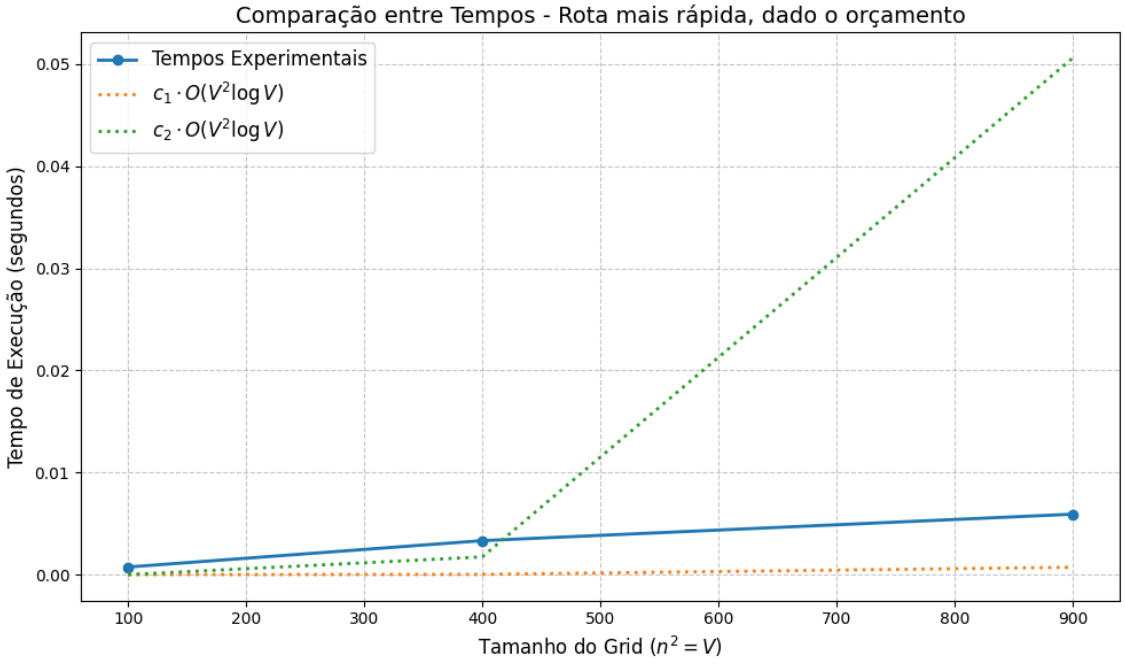

Figura 9: Comparação entre tempos experimentais e teóricos variando o número de vértices

O gráfico apresentado ilustra a comparação entre os tempos de execução experimentais e o crescimento teórico esperado da complexidade $O(V^2 \log V)$, onde $V = n^2$ representa o número total de vértices em uma grid de tamanho $n \times n$.

- A linha azul sólida com marcadores circulares representa os tempos experimentais obtidos em diferentes tamanhos de grid (10×10, 20×20, 30×30, e 40×40). Esses tempos refletem medições reais de execução do algoritmo. Como a função do tempo é cotada inferior esuperiormente por funções múltiplas de $(V^2 log V)$, o que mostra que a função tempo tem também essa complexidade

3.7 Parte 7: Encontra a aresta dado um endereço

A função findEdgeAddress encontra a aresta (edge) e o vértice (vertex) mais próximo de um número de imóvel especificado em uma rua. Seu propósito é localizar um imóvel com base em seu número, considerando as das ruas e CEPs.

3.7.1 Pseudocódigo

```
Função findEdgeAddress(street, id_zipCode, number_build, edges)
    current_distance = 0
    sorted_edges = edges

    // Ordenar as arestas com base nos IDs dos vértices
    Ordenar sorted_edges por menor ID dos vértices em ordem crescente

    Para cada edge em sorted_edges faça
        Se edge.id_street == street e edge.id_zipCode == id_zipCode então
            id_start = menor entre edge.vertex1.id e edge.vertex2.id
            id_end = maior entre edge.vertex1.id e edge.vertex2.id

            start_number = current_distance + 1
            end_number = current_distance + edge.distance

            Se number_build >= start_number E number_build <= end_number então
                relative_position = number_build - start_number
                middle_point = edge.distance / 2

                Se relative_position <= middle_point então
                    closest_vertex = edge.vertex1
                Senão
                    closest_vertex = edge.vertex2

                Retornar (edge, closest_vertex)

            current_distance = end_number
        Fim Se
    Fim Para

    Retornar (nullptr, nullptr)
Fim Função
```

Complexidade Total: O(ElogE), devido à etapa de ordenação dominar a análise