Otimização de Roteamento Urbano com Grafos

Gustavo D. Guimarães[1], **João P. S. Marques**[1], **Júlia M. Silva**[1], **Matheus C. Rocha**[1]

[1] Curso de Engenharia de Software – Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Unidade Coração Eucarístico
30.535-901 – Belo Horizonte – MG – Brazil

delfinoguimaraes1@gmail.com, jpsantanamarques2905@gmail.com

julia.medeiros1159@gmail.com, matheuscaetanorocha@gmail.com

Abstract. *This paper presents a practical application for urban route optimization using graph theory techniques and classical graph-search algorithms. We developed a solution leveraging real data from OpenStreetMap to construct georeferenced graphs and implemented the Dijkstra and A* algorithms for calculating shortest paths between user-selected points. Results demonstrate that both algorithms efficiently solve urban routing problems. We conclude that the proposed approach effectively visualizes optimized routes, enhancing practical applications in logistics and urban mobility.*

Resumo. *Este trabalho apresenta uma aplicação prática de otimização de rotas urbanas utilizando técnicas de teoria dos grafos e algoritmos clássicos de busca em grafos. Desenvolveu-se uma solução que usa dados reais extraídos do OpenStreetMap para construir grafos georreferenciados, implementando os algoritmos Dijkstra e A* para calcular caminhos mínimos entre pontos selecionados pelo usuário. Os resultados indicam que ambos os algoritmos fornecem soluções eficazes para problemas de roteamento urbano. Conclui-se que a abordagem adotada permite uma visualização clara e eficiente das rotas otimizadas, potencializando aplicações práticas em logística e mobilidade urbana.*

1. Introdução

1.1. Tema escolhido

A navegação eficiente em grandes centros urbanos é um problema recorrente que afeta diretamente a mobilidade, a logística, o transporte público e a qualidade de vida da população. Encontrar rotas mais curtas ou mais rápidas não é apenas uma comodidade, mas uma necessidade para empresas de transporte, plataformas de entrega, órgãos públicos de gestão de tráfego e usuários comuns. No contexto das cidades inteligentes, onde a integração de tecnologias visa otimizar os serviços urbanos, a solução eficiente do problema de roteamento urbano torna-se uma peça-chave para a fluidez e sustentabilidade dos centros metropolitanos [González et al. 2008, Y. Zheng 2014].

Neste projeto, propomos o desenvolvimento de uma aplicação interativa capaz de visualizar redes viárias reais e calcular rotas otimizadas entre dois pontos geográficos. A malha urbana é modelada como um grafo, onde os cruzamentos ou pontos de inflexão

são representados por nós (vértices), e os trechos de rua por arestas ponderadas de acordo com sua distância física real. Essa representação permite explorar algoritmos clássicos da ciência da computação para encontrar soluções eficientes e replicáveis em diversas regiões urbanas. A teoria dos grafos, nesse cenário, fornece uma base robusta e formal para representar e solucionar problemas de mobilidade e roteamento [Cormen et al. 2009, Delling et al. 2009].

Utilizando a linguagem de programação Python e dados abertos obtidos através da plataforma OpenStreetMap, a aplicação construída neste trabalho permite ao usuário carregar mapas urbanos no formato GeoJSON e interagir visualmente com a rede de ruas por meio da biblioteca gráfica Tkinter. Ao selecionar dois pontos quaisquer, o sistema é capaz de calcular não apenas o menor caminho, mas também uma rota alternativa viável, com base em dois algoritmos de caminhos mínimos: Dijkstra e A* (A-Star). A escolha desses algoritmos justifica-se por sua relevância teórica e aplicação prática comprovada em sistemas reais de navegação [Hart et al. 1968, Bast et al. 2016]. O resultado é uma plataforma funcional e educativa, que une teoria e prática ao permitir a visualização direta das rotas e da lógica computacional por trás de sua construção.

1.2. Algoritmos utilizados

Para o cálculo dos caminhos mínimos, foram implementados dois algoritmos clássicos e consagrados na literatura: o algoritmo de Dijkstra [Dijkstra 1959, Cormen et al. 2009] e o algoritmo A* [Hart et al. 1968, Russell and Norvig 2018]. Ambos operam sobre grafos ponderados, porém com estratégias distintas para explorar o espaço de possíveis rotas.

Algoritmo de Dijkstra: Desenvolvido por Edsger W. Dijkstra em 1959, este algoritmo resolve o problema do caminho mínimo com pesos não negativos, encontrando a rota mais curta de um vértice origem para todos os demais vértices do grafo [Dijkstra 1959]. A implementação utiliza uma fila de prioridade (heap) para selecionar iterativamente o nó com a menor distância acumulada. Em grafos esparsos, a complexidade é de $O(|E| + |V| \log |V|)$, onde $|V|$ é o número de nós e $|E|$ o número de arestas [Cormen et al. 2009]. No contexto urbano, Dijkstra garante encontrar eficientemente o caminho ótimo, mas chega a explorar regiões do grafo que podem não ser relevantes para um destino específico.

Algoritmo A*: O A* foi proposto em 1968 por Hart, Nilsson e Raphael como uma generalização heurística do algoritmo de Dijkstra [Hart et al. 1968]. Utiliza uma função de avaliação $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo do caminho já percorrido e $h(n)$ é uma heurística admissível que estima o custo restante até o destino. No caso deste trabalho, $h(n)$ corresponde à distância geodésica calculada pela fórmula de Haversine, aplicável a coordenadas geográficas. Quando $h(n)$ é consistente e nunca superestima a distância real, o A* é garantido para produzir o caminho ótimo e tende a explorar menos nós que o Dijkstra [Russell and Norvig 2018, Wikipedia contributors].

Comparação e escolha: Enquanto o Dijkstra é ideal para computar todas as rotas mínimas de uma origem a todos os destinos (espalhando a busca pelo grafo), o A* traz eficiência adicional quando se deseja uma rota entre pontos específicos – fato comprovado no desempenho observado nesta aplicação. No entanto, a implementação de ambos os algoritmos permite um comparativo interessante no ambiente gráfico, possibilitando análises sobre tempo de execução e número de nós expandidos em diferentes trechos ur-

banos.

Reconstrução dos caminhos: Tanto em Dijkstra quanto em A*, mantemos um mapa de predecessores para cada nó visitado. Após encontrar o destino, percorre-se esse mapa de trás para frente (do destino até a origem) para montar a sequência de vértices (e arestas) que formam a rota ótima. Essa abordagem é comum e eficiente, evitando armazenar todos os possíveis caminhos alternativos.

1.3. Modelagem e Diagramas

1.3.1. Diagrama de Componentes

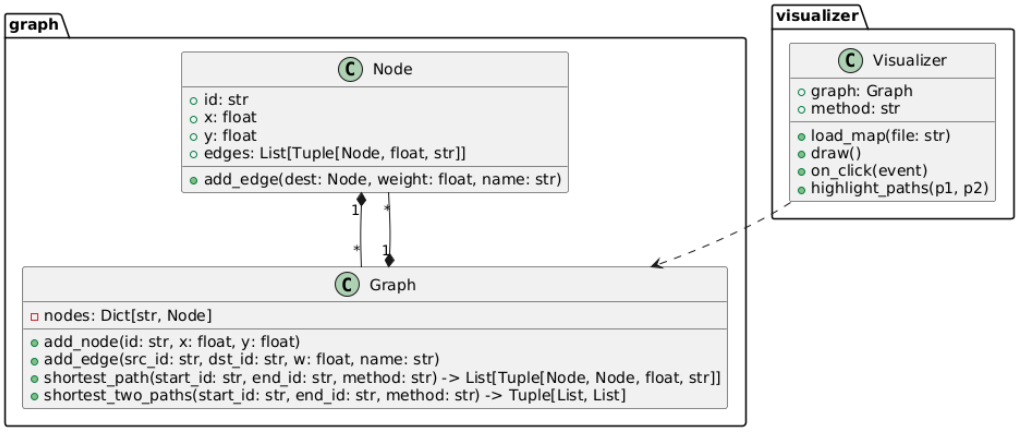

Figure 1. Diagrama de Componentes

O diagrama de componentes apresenta os principais módulos do sistema e suas interações. Ele ilustra claramente a estrutura em camadas, incluindo o parser de GeoJSON, as classes de grafo (Node e Graph) e a interface gráfica (Visualizer), evidenciando como cada componente contribui para as funcionalidades de carregamento, representação e visualização das rotas.

1.3.2. Diagrama de Sequência

O diagrama de sequência descreve o fluxo essencial da aplicação: após o usuário selecionar o ponto inicial (A) e, em seguida, o ponto final (B), o sistema solicita confirmação para o cálculo. Em seguida, a classe ‘Visualizer‘ chama o método ‘shortestTwoPaths()‘ da classe ‘Graph‘, obtendo as rotas ótima e secundária. Por fim, esses resultados são destacados no mapa e exibidos ao usuário.

1.4. Tecnologias utilizadas

Durante o desenvolvimento da aplicação, diferentes abordagens tecnológicas foram adotadas e testadas até se alcançar a solução final. Inicialmente, foi implementado um protótipo funcional utilizando JavaScript , presente na pasta `code JS`. Embora viável, essa versão demonstrou-se lenta e pouco responsiva ao processar mapas urbanos com grande volume de dados. Em função disso, optou-se por uma reestruturação total do

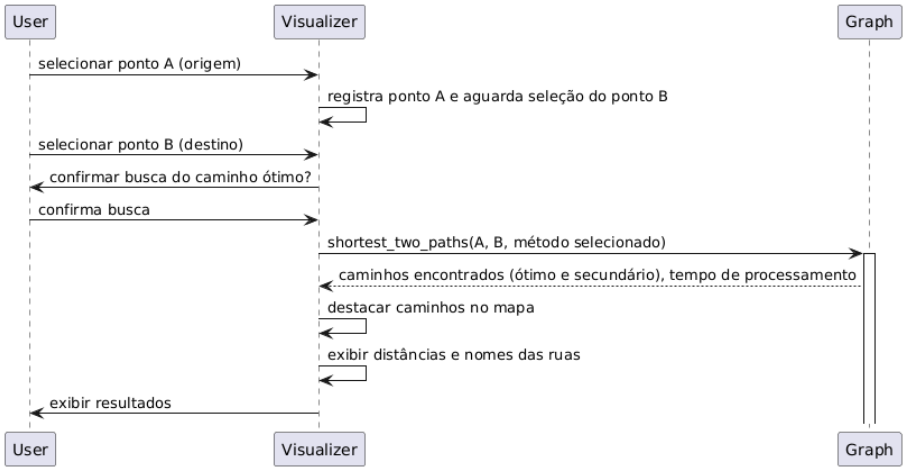

Figure 2. Diagrama de Sequência

sistema em Python, disponível na pasta `code PY`, resultando em um desempenho significativamente superior e em maior controle sobre estruturas de dados e algoritmos.

Para representação e carregamento da malha urbana, foi adotado o formato **GeoJSON** — um padrão aberto baseado em JSON, projetado para armazenar dados geoespaciais como pontos, linhas e polígonos [Internet Engineering Task Force 2016]. Por sua simplicidade, legibilidade e compatibilidade com ferramentas como OpenStreetMap, foi a escolha ideal para integração com dados reais.

Os arquivos GeoJSON foram obtidos por meio da ferramenta **Overpass Turbo**, que oferece uma interface gráfica para a linguagem de consultas da Overpass API, facilitando a extração precisa de ruas e vias urbanas de qualquer região do mundo [OpenStreetMap contributors]. Com ela, foi possível selecionar áreas específicas, como bairros ou regiões centrais, e exportar os dados diretamente no formato adequado para o processamento pelo sistema.

A **interface gráfica** da aplicação foi construída com a biblioteca **Tkinter**, que faz parte da biblioteca padrão do Python e fornece recursos suficientes para criar interfaces interativas leves, portáveis e eficazes [Shipman 2013]. Por meio dela, foi implementado um canvas gráfico para visualização do grafo, menus de interação, exibição de caminhos calculados e estatísticas relevantes ao usuário.

Para representação da arquitetura da solução e do fluxo de interação, utilizou-se a ferramenta **PlantUML**, que permite a criação de diagramas UML por meio de descrições textuais simples. À medida que a solução foi evoluindo e mudando, os diagramas também foram ajustados de acordo para sempre refletir o funcionamento da aplicação.

2. Objetivos

2.1. Objetivo geral

O objetivo principal deste trabalho é desenvolver uma ferramenta computacional que modele mapas urbanos como grafos georreferenciados e permita o cálculo e visualização interativa de rotas ótimas entre dois pontos geográficos, utilizando dados reais extraídos do OpenStreetMap.

2.2. Objetivos específicos

- Modelar redes viárias urbanas como grafos, definindo nós em cruzamentos e arestas com pesos reais baseados em distâncias.
- Implementar algoritmos clássicos de caminhos mínimos, Dijkstra e A*, para cálculo eficiente de rotas.
- Permitir ao usuário visualizar e interagir com as rotas em um mapa real, carregado via GeoJSON.
- Comparar desempenho entre os algoritmos em termos de tempo de execução e número de nós explorados.
- Fornecer uma segunda rota alternativa viável, útil para contornar congestionamentos ou bloqueios.
- Demonstrar visualmente o funcionamento da teoria de grafos aplicada a problemas reais de logística e mobilidade.

3. Justificativa

A otimização de rotas urbanas é um desafio relevante em múltiplos contextos, incluindo logística de entrega, planejamento de transporte público e gerenciamento de tráfego em cidades inteligentes [Gonçalo S. et al. 2021, Dagli et al. 2016]. A aplicação da teoria dos grafos permite modelar de forma eficaz redes viárias e utilizar algoritmos bem estabelecidos para solucionar problemas de roteamento [Guze 2018]. O uso de dados reais do OpenStreetMap, acessados via Overpass Turbo, aumenta a relevância prática do sistema, tornando-o útil para cenários reais. Além disso, o trabalho contribui para a formação acadêmica, ao proporcionar experiência em manipulação de dados geográficos, implementação de algoritmos de grafos e design de interfaces gráficas. Em um contexto educacional, permite que futuros desenvolvedores e pesquisadores compreendam de forma prática como a teoria de grafos pode ser aplicada em problemas cotidianos de mobilidade urbana.

4. Desenvolvimento

Esta seção descreve em detalhes o processo de implementação do sistema de otimização de rotas urbanas desenvolvido no contexto deste trabalho. São apresentados os métodos de leitura e pré-processamento dos dados geográficos, a modelagem do grafo, a implementação dos algoritmos de menor caminho, o funcionamento da interface gráfica e os principais testes e avaliações realizados.

4.1. Leitura e Pré-processamento dos Dados

O sistema foi projetado para utilizar mapas urbanos reais obtidos no formato GeoJSON, um padrão amplamente utilizado para representação de dados geográficos. Para alimentar o sistema com informações fidedignas da malha urbana de Belo Horizonte, foram extraídos dados do OpenStreetMap através da ferramenta Overpass Turbo, possibilitando a seleção personalizada de regiões e tipos de vias.

Após a seleção do arquivo GeoJSON, o sistema realiza o processamento das informações geográficas para transformá-las em uma estrutura de grafo. Cada elemento do tipo LineString ou Polygon presente no arquivo é interpretado como uma rua, rotatória ou praça, e seus pontos extremos (coordenadas de latitude e longitude) são convertidos em

nós do grafo. As conexões entre esses pontos são registradas como arestas, cujos pesos correspondem à distância real entre os pontos, calculada por meio da fórmula de Haversine. Essa abordagem garante maior precisão na modelagem das distâncias, considerando a curvatura da Terra.

O pré-processamento dos dados, portanto, envolve:

- Leitura e parsing do arquivo GeoJSON;
- Identificação e conversão dos elementos geográficos em nós e arestas do grafo;
- Cálculo das distâncias reais entre pontos para atribuição dos pesos das arestas.

Esse fluxo automatizado permite ao sistema receber diferentes mapas urbanos e adaptá-los facilmente ao contexto da otimização de rotas, sem a necessidade de ajustes manuais.

O procedimento de leitura e pré-processamento dos dados pode ser observado na Listagem 1.

```
        # Para cada feature no GeoJSON, extrai ruas e pontos
        for feat in data['features']:
            geometry = feat.get('geometry')
            if not geometry:
                continue

            street = feat['properties'].get('name', '')
            geom_type = geometry.get('type')
            coords = geometry.get('coordinates')

            # Lista que conter  os pontos a serem processados
            points_to_process = []

            # Trata tanto LineString quanto Polygon
            if geom_type == 'LineString':
                points_to_process = coords
            elif geom_type == 'Polygon':
                # Para pol gonos, as coordenadas  s  o uma lista de
    an is.
                # Usamos apenas o primeiro anel (o contorno externo).
                if coords and len(coords) > 0:
                    points_to_process = coords[0]

            # Se  n  o  for um tipo de geometria que sabemos tratar,
    pulamos para a  pr xima  feature
            if not points_to_process:
                continue

            # Agora o processamento    seguro, pois points_to_process
       uma lista simples de pontos
            for lon, lat in points_to_process:
                self.graph.add_node(str((lon, lat)), lon, lat)

            # Cria arestas entre pontos consecutivos
            for i in range(len(points_to_process) - 1):
                u_lon, u_lat = points_to_process[i]
                v_lon, v_lat = points_to_process[i+1]
                w = haversine_distance(u_lat, u_lon, v_lat, v_lon)
```

```
36                 self.graph.add_edge(str((u_lon, u_lat)), str((v_lon,
   v_lat)), w, street)
```

Listing 1. Leitura e pré-processamento do GeoJSON

4.2. Modelagem do Grafo

O grafo urbano utilizado no sistema é modelado de forma orientada a objetos, garantindo flexibilidade e clareza no processamento das rotas. Cada ponto de interesse extraído do arquivo GeoJSON é representado por uma instância da classe Node, contendo um identificador único e suas coordenadas geográficas (longitude e latitude). As ruas, rotatórias ou conexões entre pontos são modeladas como arestas, armazenadas em uma lista dentro de cada nó, contendo o nó de destino, o peso (distância real entre os pontos) e, opcionalmente, o nome da rua.

A estrutura do grafo permite tanto a busca eficiente de vizinhos quanto a execução dos algoritmos de menor caminho, além de facilitar a visualização dos caminhos percorridos.

A modelagem orientada a objetos pode ser vista na Listagem 2.

```
1  class Node:
2      """
3      Representa um  n   do grafo, com identificador, coordenadas e lista
      de arestas.
4      """
5      def __init__(self, id: str, x: float, y: float):
6          self.id = id
7          self.x = x  # Longitude
8          self.y = y  # Latitude
9          self.edges: List[Tuple['Node', float, str]] = []
10
11     def add_edge(self, dest: 'Node', weight: float, name: str):
12         """
13         Adiciona uma aresta ligando este  n   a outro  n   do grafo.
14         """
15         self.edges.append((dest, weight, name))
16
17 class Graph:
18     """
19     Estrutura principal do grafo, armazena os  n  s  e permite executar
      algoritmos de caminhos  m nimos.
20     """
21     def __init__(self):
22         self.nodes: Dict[str, Node] = {}
23
24     def add_node(self, id: str, x: float, y: float):
25         """
26         Adiciona um novo  n   ao grafo.
27         """
28         if id not in self.nodes:
29             self.nodes[id] = Node(id, x, y)
30
31     def add_edge(self, src_id: str, dst_id: str, w: float, name: str =
      ""):
32         """
```

```
        Adiciona uma aresta bidirecional entre dois  n  s  do grafo.
        """
        if src_id in self.nodes and dst_id in self.nodes:
            src = self.nodes[src_id]
            dst = self.nodes[dst_id]
            src.add_edge(dst, w, name)
            dst.add_edge(src, w, name)
```

Listing 2. Modelagem orientada a objetos dos nós e arestas do grafo.

4.3. Implementação dos Algoritmos

Para a otimização de rotas, o sistema implementa dois algoritmos clássicos de caminhos mínimos em grafos: **Dijkstra** e **A*** (A-star). Ambos são utilizados para encontrar a rota de menor custo entre dois pontos de interesse no mapa urbano modelado.

O algoritmo de Dijkstra, ilustrado na Listagem 3, foi escolhido por sua robustez e simplicidade, garantindo a obtenção do caminho mais curto em grafos com pesos não negativos. Sua implementação utiliza uma fila de prioridade (*min-heap*) para selecionar o próximo nó a ser explorado com o menor custo acumulado.

O algoritmo A*, apresentado na Listagem 4, diferencia-se por incorporar uma heurística que estima o custo restante até o destino, acelerando a busca por caminhos em mapas de grande escala. No sistema desenvolvido, a heurística utilizada é a distância de Haversine entre o nó corrente e o destino, cuja implementação está na Listagem 5.

A escolha entre os algoritmos pode ser feita pelo usuário por meio da interface gráfica, permitindo comparar o desempenho e os caminhos encontrados por cada abordagem.

```
def _dijkstra(self, start_id: str) -> Tuple[Dict[str, float], Dict[str,
    str]]:
        """
        Implementa o algoritmo de Dijkstra para encontrar o menor
   caminho a partir de um  n   de origem
        para todos os outros  n  s  do grafo.
        Retorna: dicion rio de dist ncias  m nimas  e dicion rio de
   predecessores.
        """
        dist = {nid: math.inf for nid in self.nodes}
        prev: Dict[str, str] = {}
        dist[start_id] = 0

        # Fila de prioridade: (dist ncia acumulada, id do  n  )
        pq = [(0, start_id)]

        while pq:
            d, u_id = heapq.heappop(pq)

            # Se  j   encontramos um caminho menor, ignoramos este
            if d > dist[u_id]:
                continue

            u_node = self.nodes[u_id]
            for v_node, w, _ in u_node.edges:
                if dist[u_id] + w < dist[v_node.id]:
```

```
                    dist[v_node.id] = dist[u_id] + w
                    prev[v_node.id] = u_id
                    heapq.heappush(pq, (dist[v_node.id], v_node.id))

        return dist, prev
```

Listing 3. Implementação do algoritmo de Dijkstra.

```
def _astar_prev(self, start_id: str, end_id: str) -> Dict[str, str]:
        """
        Implementa o algoritmo A* para encontrar o menor caminho entre dois n s do grafo.
        Usa a heur stica de Haversine para guiar a busca.
        Retorna um dicion rio de predecessores para reconstru o do caminho.
        """
        open_set = {start_id}
        came_from: Dict[str, str] = {}

        g_score = {nid: math.inf for nid in self.nodes}
        g_score[start_id] = 0

        f_score = {nid: math.inf for nid in self.nodes}
        f_score[start_id] = self._heuristic(start_id, end_id)

        while open_set:
            # Seleciona o  n   com menor f_score
            current = min(open_set, key=lambda nid: f_score[nid])

            if current == end_id:
                return came_from

            open_set.remove(current)

            for neighbor, w, _ in self.nodes[current].edges:
                tentative_g = g_score[current] + w
                if tentative_g < g_score[neighbor.id]:
                    came_from[neighbor.id] = current
                    g_score[neighbor.id] = tentative_g
                    f_score[neighbor.id] = tentative_g + self._heuristic(neighbor.id, end_id)
                    if neighbor.id not in open_set:
                        open_set.add(neighbor.id)

        return came_from
```

Listing 4. Implementação do algoritmo A*.

A fórmula de Haversine, apresentada na Listagem 5, é utilizada tanto para atribuir pesos reais às arestas quanto como heurística admissível no algoritmo A*. Essa fórmula calcula a distância mais curta entre dois pontos sobre a superfície de uma esfera, considerando a curvatura da Terra, sendo adequada para aplicações em mapas urbanos.

```
def haversine_distance(lat1: float, lon1: float, lat2: float, lon2: float) -> float:
    """
```

```
    Calcula a dist ncia em metros entre duas coordenadas geogr ficas
    (latitude/longitude)
    utilizando a f rmula de Haversine, que considera a curvatura da
    Terra.
    """
    d_lat = math.radians(lat2 - lat1)
    d_lon = math.radians(lon2 - lon1)
    rad_lat1 = math.radians(lat1)
    rad_lat2 = math.radians(lat2)

    a = (math.sin(d_lat / 2) ** 2) + (math.cos(rad_lat1) * math.cos(
    rad_lat2) * math.sin(d_lon / 2) ** 2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    return RAIO_TERRA_M * c
```

Listing 5. Implementação da fórmula de Haversine para cálculo de distância geográfica.

A implementação detalhada dos algoritmos garante a flexibilidade do sistema, permitindo inclusive a busca do segundo melhor caminho por meio da remoção temporária das arestas do caminho ótimo encontrado.

4.4. Interface Gráfica

Para proporcionar uma experiência interativa ao usuário, o sistema conta com uma interface gráfica desenvolvida em Python utilizando o framework Tkinter. Logo ao iniciar o sistema, o usuário é solicitado a selecionar um arquivo de mapa no formato GeoJSON, que é então processado e exibido visualmente.

A interface é composta por duas áreas principais: o *canvas* de visualização e a barra lateral de controles. No canvas, é apresentado o grafo correspondente ao mapa urbano carregado, com nós representando pontos geográficos e arestas representando ruas e conexões. O usuário pode selecionar, com cliques do mouse, o ponto de partida e o ponto de chegada da rota desejada. Após a seleção, o sistema calcula e destaca, no próprio canvas, o menor caminho e o segundo melhor caminho entre os pontos escolhidos.

A barra lateral oferece funcionalidades como:

- Exibição do status do sistema e instruções de uso;
- Exibição do número de nós e arestas carregados;
- Seleção do algoritmo de roteamento (Dijkstra ou A*);
- Visualização dos detalhes das rotas encontradas (distância, ruas percorridas);
- Indicação do tempo de execução do algoritmo;
- Botão para redefinir seleções de pontos e caminhos.

A interface ainda oferece recursos de navegação, como zoom (utilizando a roda do mouse) e movimentação do mapa (pan) através do arrasto com o botão direito do mouse, facilitando a visualização de mapas urbanos mais extensos.

A Figura 3 apresenta a interface gráfica do sistema após o cálculo dos caminhos entre dois pontos na malha urbana.

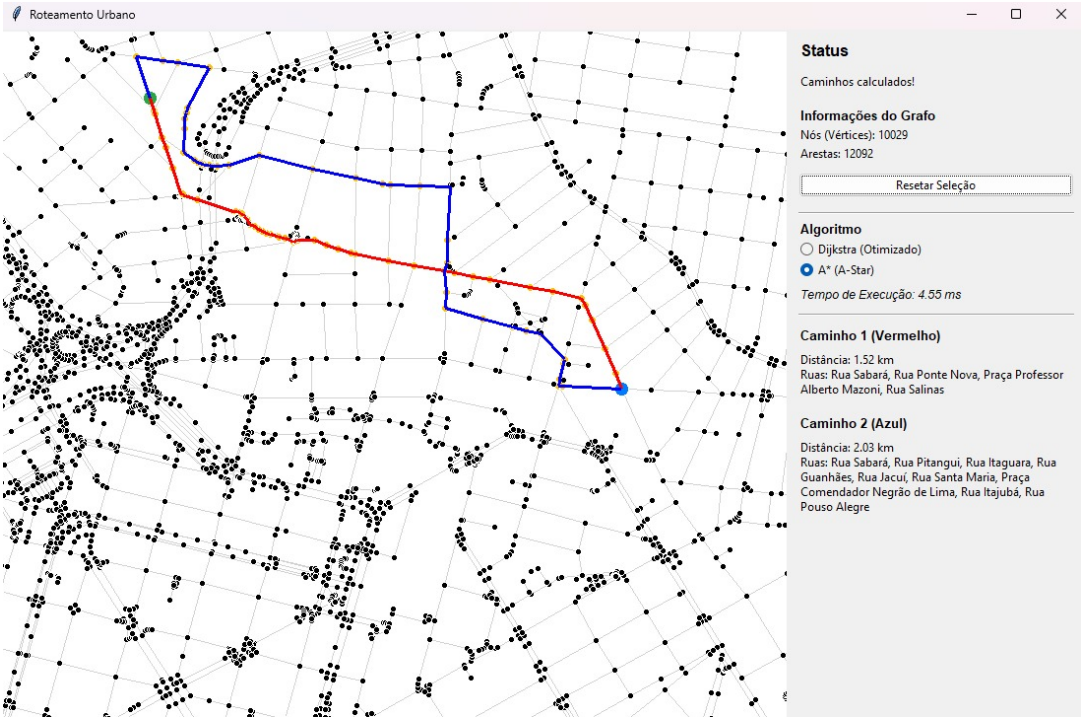

Figure 3. Interface gráfica do sistema exibindo o grafo urbano de Belo Horizonte, com destaque para o menor caminho (vermelho) e o segundo melhor caminho (azul) encontrados pelo algoritmo A*.

4.5. Desempenho e Testes

A avaliação do desempenho do sistema foi realizada em dois computadores diferentes, cujas configurações estão apresentadas na Tabela 1. Os testes buscaram analisar o tempo de execução dos algoritmos Dijkstra e A* ao calcular rotas em mapas urbanos reais de Belo Horizonte, considerando diferentes recortes de tamanho (raios de 2 km, 10 km e a cidade completa).

Table 1. Configurações dos computadores utilizados nos testes.

Computador	Processador	Memória RAM	Sistema Operacional
PC 1	13th Gen Intel Core i5	16 GB	Windows 11
PC 2	AMD Ryzen 5 5600 6-Core	16 GB	Windows 11

Os resultados dos testes de desempenho estão sumarizados na Tabela 2. Para os recortes menores (raio de 2 km), ambos os algoritmos apresentaram tempos de execução inferiores a 20 ms, permitindo uso interativo e rápido na interface. Ao aumentar o tamanho do mapa para um raio de 10 km, o tempo de processamento cresceu de forma significativa para ambos os algoritmos, mas especialmente para o A*, que ficou bem mais lento do que o Dijkstra em ambos os casos. Para a cidade inteira, o sistema não conseguiu processar o arquivo, travando a interface em ambos os computadores avaliados.

Table 2. Tempos de execução (em milissegundos) dos algoritmos Dijkstra e A* em diferentes computadores e tamanhos de mapas.

Mapa	Algoritmo	PC 1	PC 2
2*BH (raio 2 km)	Dijkstra	19,80	18,84
	A*	12,79	17,17
2*BH (raio 10 km)	Dijkstra	298,59	378,58
	A*	2744,29	3343,40
2*BH Completo	Dijkstra	Não processou	
	A*	Não processou	

Analisando os resultados, observa-se que para mapas pequenos o A* chegou a ser um pouco mais rápido que o Dijkstra, como era esperado. No entanto, para mapas maiores, o tempo de execução do A* aumentou de forma muito mais acentuada, tornando-se mais lento que o Dijkstra. Esse comportamento não é o esperado teoricamente, pois o A* deveria ser mais eficiente graças ao uso da heurística. Porém, na implementação adotada neste projeto, o conjunto aberto (*open set*) do A* foi implementado como um conjunto comum do Python, o que exige uma busca linear pelo menor *f-score* em cada iteração. Em mapas grandes, esse processo pode ser mais custoso que a fila de prioridade (*heap*) utilizada no Dijkstra, explicando o desempenho inferior do A* nos testes mais pesados.

De modo geral, o sistema mostrou-se rápido e responsivo para recortes de bairros e regiões menores, sendo adequado para aplicações didáticas ou análises locais. Em mapas urbanos completos, limitações de hardware e de implementação passaram a impactar diretamente o desempenho, tornando inviável a execução.

4.6. Destaques e Diferenciais

O sistema desenvolvido apresenta características que ampliam seu potencial didático e prático no contexto da otimização de rotas urbanas:

- **Aceita mapas urbanos de qualquer cidade:** O sistema aceita mapas urbanos de qualquer cidade, desde que sejam previamente baixados do OpenStreetMap utilizando ferramentas como o Overpass Turbo. Dessa forma, é possível aplicar a solução em diferentes regiões, sem necessidade de adaptações específicas no código.
- **Algoritmos adaptados a partir de referências:** Para a implementação dos algoritmos de Dijkstra e A*, recorremos a exemplos e discussões em fóruns e sites especializados. A partir dessas referências, adaptamos o código para atender ao nosso contexto, priorizando clareza e entendimento da lógica, sem depender de bibliotecas prontas de grafos.
- **Distâncias geográficas realistas:** Utilizamos a fórmula de Haversine para calcular distâncias reais entre os pontos, deixando as rotas encontradas mais fiéis à malha urbana.
- **Interface gráfica intuitiva:** O usuário pode carregar mapas, escolher pontos de partida e chegada, alternar entre algoritmos e visualizar as rotas diretamente na tela, o que facilita a compreensão do funcionamento do sistema.

- **Rotas alternativas:** Além do menor caminho, o sistema também mostra uma segunda opção de rota, caso exista, ampliando as possibilidades de análise.
- **Testes variados:** Rodamos o sistema em diferentes computadores e recortes de mapas, para garantir que funcionasse tanto em bairros pequenos quanto em áreas maiores, identificando limites práticos para arquivos muito grandes.
- **Código comentado e organizado:** Priorizamos escrever o código de forma clara, com comentários explicando as funções principais, para facilitar a manutenção ou futuras melhorias.

No geral, acreditamos que o sistema é uma base sólida para outros projetos envolvendo mobilidade urbana, podendo ser expandido para novos algoritmos ou funcionalidades conforme a necessidade.

5. Conclusão

Neste trabalho, foi possível atingir os objetivos propostos, oferecendo uma ferramenta capaz de modelar mapas urbanos como grafos georreferenciados, implementar algoritmos de caminhos mínimos (Dijkstra e A*), e permitir a visualização interativa de rotas sobre mapas reais obtidos via OpenStreetMap. Além disso, agregou-se valor ao sistema ao disponibilizar uma segunda rota alternativa viável, fundamentada em uma rotina específica de cálculo de "dois melhores caminhos". A aplicação provou ser útil tanto para fins práticos de roteamento quanto para propósitos pedagógicos, demonstrando o poder da teoria de grafos em cenários reais.

Ainda assim, algumas limitações foram identificadas. Mapas muito grandes — por exemplo, ao importar toda a região metropolitana de Belo Horizonte — podem levar ao esgotamento de memória e queda de desempenho ou até falhas no programa. Outro desafio tem relação com a natureza dinâmica das redes urbanas, afetadas por congestionamentos ou bloqueios: os algoritmos estáticos implementados não se adaptam em tempo real a essas variações, enquanto soluções dinâmicas exigem infraestrutura de dados em tempo real e processamento contínuo [Voloch et al. 2019, Mohamed H. El-Ela 2024]. Além disso, como a heurística do A* foi baseada apenas em distância geodésica, não incorporou variáveis como velocidade média de tráfego ou semáforos, o que pode gerar discrepâncias em contextos urbanos reais.

Visando melhorias futuras, considera-se promissora a extensão do trabalho para manipulação e visualização filtrada das ruas (por tipo ou hierarquia de via), integração de dados em tempo real (por exemplo, do trânsito) e adaptação dos algoritmos para respostas dinâmicas. Outra frente relevante seria a adoção de otimizações estruturais para suportar grafos maiores — como particionamento, compressão ou algoritmos paralelos — facilitando a análise de regiões metropolitanas inteiras [Voloch et al. 2019]. Por fim, uma possibilidade de pesquisa futura é implementar heurísticas híbridas ou algoritmos mais sofisticados (ex.: A* dinâmico, BC dinâmico) para monitoramento e roteamento mais adaptável e realista utilizando fontes de dados compatíveis com os princípios de cidades inteligentes.

References

Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2016). Route planning in transportation networks. *Algorithm Engineering*, 18:1–48.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

Dagli, C. H. et al. (2016). Applied graph theory to real smart city logistic problems. In *Procedia Computer Science*, volume 95, pages 40–47.

Delling, D., Sanders, P., Schultes, D., and Wagner, D. (2009). Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139. Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

González, M. C., Hidalgo, C. A., and Barabási, A.-L. (2008). Understanding individual human mobility patterns. In *Nature*, volume 453, pages 779–782.

Gonçalo S., R. B. et al. (2021). Smart cities and urban logistics: a systematic review of the literature. *Europa XXI*. citado em 2025-06-08.

Guze, S. (2018). Review of methods and algorithms for modelling transportation networks based on graph theory. *Scientific Journal of Gdynia Maritime University*, 107.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume 4, pages 100–107.

Internet Engineering Task Force (2016). The geojson format (rfc 7946). `https://datatracker.ietf.org/doc/html/rfc7946`.

Mohamed H. El-Ela, A. H. F. (2024). Deep heuristic learning for real-time urban pathfinding. *ArXiv*.

OpenStreetMap contributors. Overpass turbo: web-based query tool for openstreetmap. `https://overpass-turbo.eu`. accessed 2025-06-08.

Russell, S. J. and Norvig, P. (2018). *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition.

Shipman, J. W. (2013). *Tkinter 8.5 reference: a GUI for Python*. New Mexico Tech.

Voloch, N., Bloch, N. V., and Zadok, Y. (2019). Managing large distributed dynamic graphs for smart city network applications. *Applied Network Science*, 4:130.

Wikipedia contributors. A* algorithm. `https://en.wikipedia.org/wiki/A*_search_algorithm`. accessed 2025-06-08.

Y. Zheng, F. Liu, H.-P. H. (2014). U-air: When urban air quality inference meets big data. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1436–1444.