Relatório Técnico: Otimização de Rotas Urbanas com Base em Mapas

Integrantes: Isadora Maria de Araujo Cathalat, Leticia Amaral Figueiredo, Thiago Borges Laass

1. Tema

Otimização de rotas urbanas com base em mapas.

---

2. Descrição do Problema

Em ambientes urbanos, determinar o caminho mais eficiente entre dois pontos exige a análise de uma grande quantidade de rotas. Isso se torna impraticável manualmente, especialmente ao considerar a curvatura real das vias.

---

3. Justificativa da Modelagem

A modelagem do sistema foi orientada pelos seguintes critérios:

- **Eficiência computacional**
- **Fidelidade geográfica**
- **Compatibilidade com dados públicos (OSM)**

Representação:

- **Grafo direcionado e ponderado**
- **Vértices**: pontos ao longo das vias
- **Arestas**: conexões entre vértices consecutivos, com pesos em metros

Optou-se pelo algoritmo de **Dijkstra** por sua exatidão e compatibilidade com grafos de pesos não negativos.

---

4. Plano de Desenvolvimento

4.1 Objetivo

Desenvolver um modelo baseado em teoria dos grafos para encontrar o caminho mais curto considerando o traçado real da malha urbana.

4.2 Levantamento de Requisitos

Funcionais:

- Seleção da cidade
- Construção do grafo
- Cálculo do caminho mínimo
- Visualização no mapa

Não funcionais:

- Interface responsiva
- Backend modular
- Código limpo e documentado
- Facilidade de manutenção

4.3 Arquitetura e Design

- **Frontend e Backend separados**
- **Tecnologias:**
    - Flask (API backend)

    - Leaflet.js (mapas)
    - Geopy (geocodificação)
    - OpenStreetMap (fonte de dados)

Módulos:

- `app.py`: controle da aplicação
- `data.py`: leitura de dados
- `graph.py`: grafo e algoritmos
- `home.html`, `style.css`, `script.js`: interface

4.4 Implementação

Etapas:

- Interface e integração com mapas
- Upload e leitura do JSON (OSM)
- Construção do grafo
- Algoritmo de Dijkstra
- Localização de nós
- Exibição de rotas
- Tratamento de exceções

---

5. Levantamento de Dados

5.1 Fonte de Dados

- Dados do **OpenStreetMap (OSM)** via **Overpass API**
- Elementos:
    - `node`: ponto geográfico
    - `way`: via composta por `nodes`

5.2 Ferramentas e Bibliotecas

- `geopy`: geocodificação
- `heapq`: fila de prioridade
- `math`: cálculo de distâncias
- `defaultdict`: estrutura do grafo
- `re`: tratamento de texto
- `Data`: leitura de arquivos JSON

---

6. Modelagem do Grafo

6.1 Representação

- **Dicionário de adjacência**
    - Chave: ID do nó
    - Valor: lista de tuplas (vizinho, distância)

6.2 Construção

- Mapear nós e coordenadas
- Criar arestas entre nós consecutivos nas vias

- Arestas duplas quando a via **não for mão única**
- Distâncias calculadas com **fórmula de Haversine**

6.3 Distância Geográfica

- A função `haversine(a, b)` calcula a distância real entre dois pontos na Terra.

---

7. Algoritmo de Caminho Mínimo

7.1 Dijkstra

- Inicializa distâncias com infinito
- Fila de prioridade (`heapq`)
- Atualização iterativa das menores distâncias
- Reconstrução do caminho ao final

7.2 Validação de Nós

- Verifica se os nós de origem e destino existem
- Se não existirem, retorna caminho vazio e distância infinita

---

8. Localização de Nós

8.1 Encontrar o Nó Mais Próximo

- `nearest_node()` localiza o nó mais próximo a uma coordenada dada
- Utiliza **distância euclidiana** para eficiência

8.2 Geocodificação

- `Nominatim` via `geopy` permite converter endereços em coordenadas

---

9. Execução e Resultados

9.1 Entrada

- `origem_coords` e `destino_coords`: coordenadas geográficas
- `filename`: nome do arquivo JSON com dados OSM

9.2 Saída

- Lista de IDs dos nós no caminho
- Distância total em metros
- Lista de arestas com:
    - Origem (coordenadas)
    - Destino (coordenadas)
    - Peso (distância em metros)

---

10. Testes e Validação

10.1 Testes Realizados

- Testes manuais com diferentes cidades
- Casos de erro: caminho inexistente, arquivo inválido, etc.

10.2 Resultados Esperados

- Caminho correto e distância precisa
- Visualização no mapa
- Comunicação eficaz entre backend e frontend

10.3 Testes Futuros

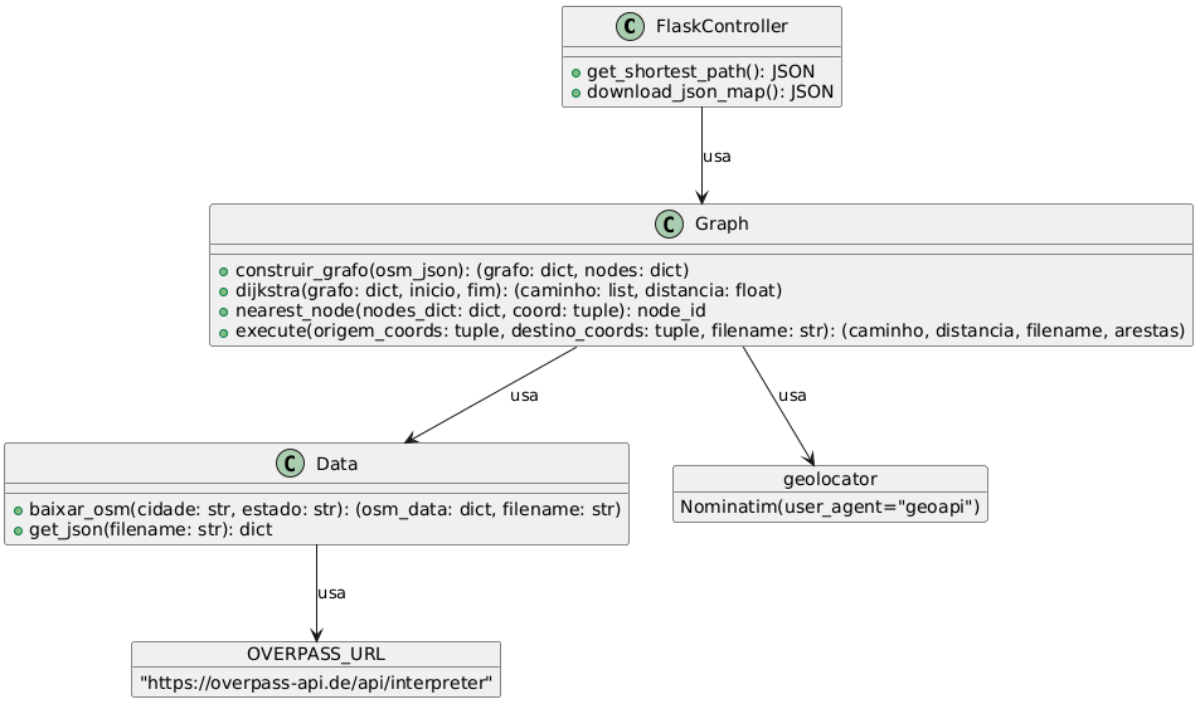
- Testes automatizados com `pytest`
- Testes de desempenho com grafos grandes

---

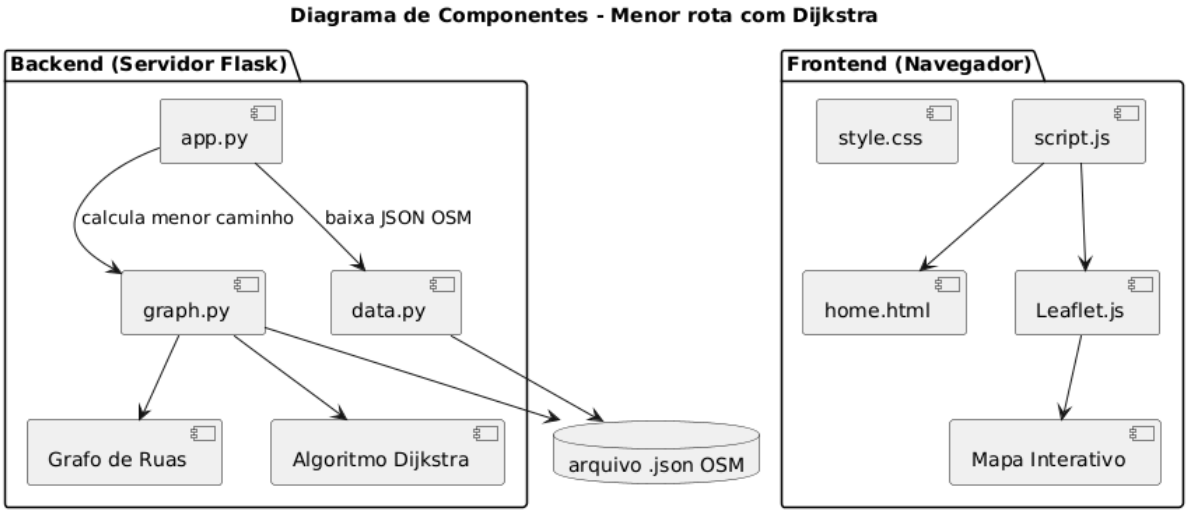
11. Diagramas

11.1 Casos de usuário

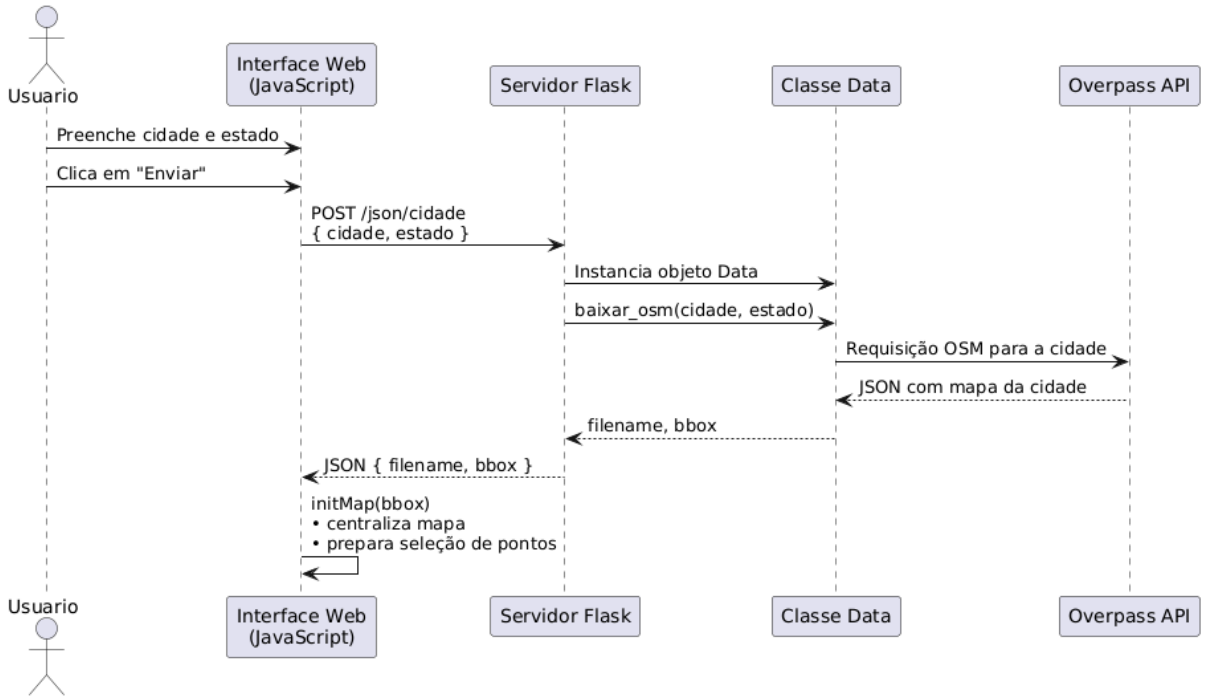
11.2 Diagrama de Classes



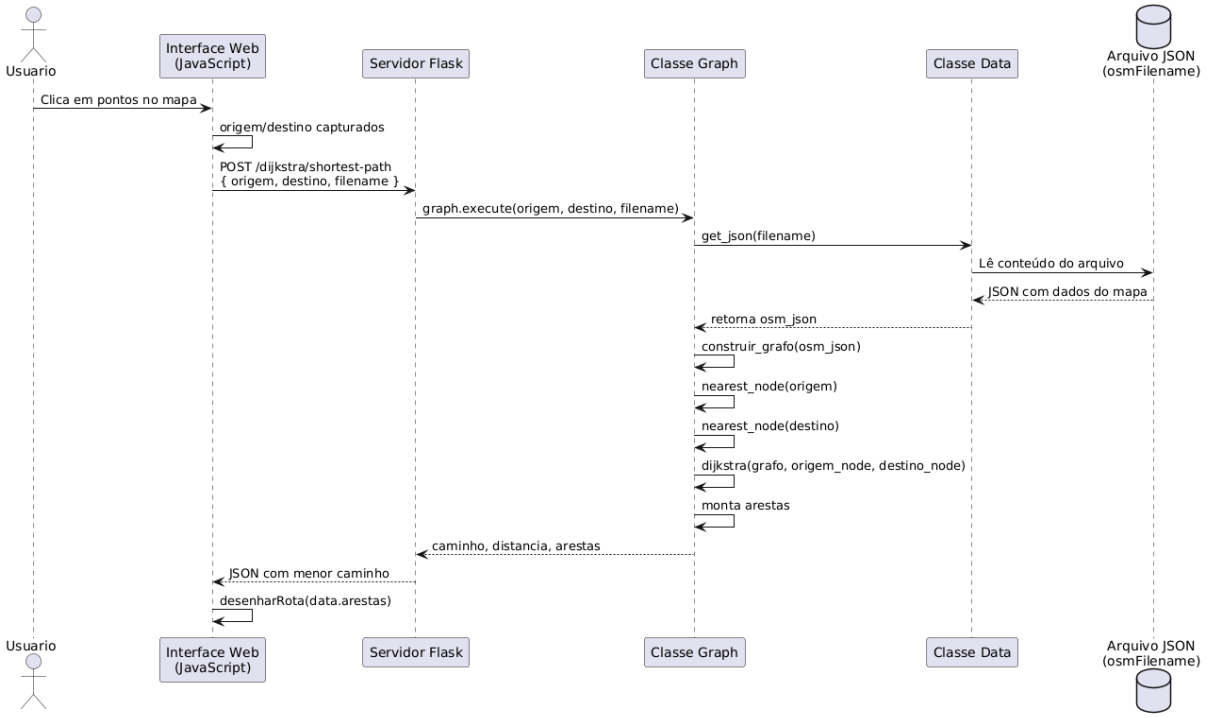
11.3 Diagrama de Componentes



11.4 Diagrama de Sequência 1



11.5 Diagrama de Sequência 2



---

12. Manutenção e Evolução

Melhorias previstas:

- Otimização da estrutura do grafo
- Dockerização
- Logs e monitoramento

---

13. Ferramentas e Organização

- **Ambiente virtual Python (`venv`)**
- **Dependências listadas em `requirements.txt`**
- **Controle de versão com Git**
- **Estrutura de diretórios:**
    - `backend/`: código Python
    - `frontend/`: HTML, CSS, JS