



FACULDADE CESAR SCHOOL
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Análise de Redes de Bairros do Recife: Aplicação de Teoria de Grafos

Júlia Sales
Miguel Becker
Thiago Queiroz

Recife
Novembro/2025

Sumário

1. Resumo	4
2. Introdução	5
2.1. Objetivos	5
2.2. Escopo	5
3. Fundamentação Teórica	6
3.1. Definições Básicas	6
3.2. Algoritmos Implementados	6
3.2.1. Busca em Largura (BFS)	6
3.2.2. Busca em Profundidade (DFS)	6
3.2.3. Algoritmo de Dijkstra	6
3.2.4. Algoritmo de Bellman-Ford	6
3.3. Métricas de Rede	7
3.3.1. Densidade	7
3.3.2. Grau Médio	7
3.3.3. Ego-Network	7
4. Metodologia	8
4.1. Estrutura de Dados	8
4.2. Sistema de Cálculo de Pesos	8
4.2.1. Componentes do Peso	8
4.3. Processamento de Dados	9
4.3.1. Obtenção das Interconexões	9
4.3.2. Etapa 1: Normalização dos Bairros	10
4.3.3. Etapa 2: Construção do Grafo	10
4.3.4. Etapa 3: Cálculo de Métricas	10
4.4. Análise de Caminhos Mínimos	11

4.5. Visualizações	11
5. Resultados	12
5.1. Características da Rede	12
5.2. Distribuição de Pesos	12
5.3. Caso de Estudo: Nova Descoberta → Boa Viagem	12
5.4. Validação dos Algoritmos	13
5.4.1. Testes de BFS (6 casos)	13
5.4.2. Testes de DFS (6 casos)	13
5.4.3. Testes de Dijkstra (7 casos)	14
5.4.4. Testes de Bellman-Ford (7 casos)	14
5.5. Visualizações Geradas	14
6. Parte 2: Dataset Maior e Comparação de Algoritmos	15
6.1. Descrição do Dataset	15
6.1.1. Distribuição de Graus	15
6.2. Experimentos BFS/DFS	15
6.2.1. Resultados BFS	16
6.2.2. Resultados DFS	16
6.3. Experimentos Dijkstra	16
6.4. Experimentos Bellman-Ford	17
6.4.1. Caso 1: Pesos Negativos sem Ciclo Negativo	17
6.4.2. Caso 2: Grafo com Ciclo Negativo	17
6.5. Comparação de Desempenho	18
6.5.1. Análise Crítica	18
6.5.2. Limites do Design de Pesos	18
7. Discussão	19
7.1. Interpretação dos Resultados	19
7.2. Sistema de Pesos	19
7.3. Eficiência dos Algoritmos	19
7.4. Limitações	19
7.4.1. Limitações da Coleta de Dados	20
7.4.2. Limitações do Sistema de Pesos	20
7.4.3. Limitações do Modelo	20

8. Código-Fonte	21
8.1. Módulo Principal: Graph	21
8.2. Algoritmos de Busca (BFS e DFS)	22
8.3. Algoritmo de Dijkstra	23
8.4. Algoritmo de Bellman-Ford	23
8.5. Cálculo de Pesos	24
9. Conclusão	26
9.1. Parte 1: Rede de Bairros do Recife	26
9.2. Parte 2: Dataset de Larga Escala	26
9.3. Lições Aprendidas	26

1. Resumo

Este trabalho apresenta uma análise computacional da rede de bairros da cidade do Recife sob a perspectiva da Teoria de Grafos. A aplicação foi desenvolvida em Python e utiliza estruturas de dados eficientes (listas de adjacência) para representar a malha urbana, permitindo a aplicação de algoritmos clássicos de grafos como BFS, DFS, Dijkstra e Bellman-Ford.

O sistema implementa funcionalidades completas de análise de redes, incluindo cálculo de métricas topológicas (densidade, grau médio, componentes conexos), determinação de caminhos mínimos ponderados entre localidades, e geração de visualizações interativas. Os pesos das arestas foram calculados considerando características reais das vias, como tipo de pavimentação e presença de obstáculos.

A validação do sistema foi realizada através de uma suíte de 26 testes unitários, confirmando a correção dos algoritmos implementados. Os resultados demonstram a viabilidade da aplicação de grafos para modelagem e análise de redes urbanas reais.

2. Introdução

A Teoria de Grafos desempenha papel fundamental na modelagem e análise de redes em diversos domínios, incluindo redes de transporte, sistemas de comunicação e planejamento urbano. Este projeto aplica conceitos fundamentais de grafos para analisar a estrutura de conectividade entre os bairros do Recife.

2.1. Objetivos

Os principais objetivos deste trabalho incluem:

- Modelar a rede de bairros do Recife como um grafo não-direcionado ponderado
- Implementar algoritmos clássicos de busca e caminho mínimo
- Calcular métricas topológicas da rede (densidade, grau médio, ego-networks)
- Desenvolver sistema de pesos baseado em características reais das vias
- Determinar rotas ótimas entre localidades específicas
- Gerar visualizações interativas para análise exploratória
- Validar a implementação através de testes automatizados

2.2. Escopo

O projeto abrange 94 bairros da cidade do Recife, organizados em 6 microrregiões, com 244 conexões viárias mapeadas. Cada conexão (aresta) possui atributos como nome do logradouro, tipo de via, pavimentação e peso calculado.

3. Fundamentação Teórica

3.1. Definições Básicas

Um grafo $G = (V, E)$ é definido por um conjunto de vértices V e um conjunto de arestas $E \subseteq V \times V$. No contexto deste projeto:

- **Vértices (V):** Representam os bairros do Recife
- **Arestas (E):** Representam conexões viárias entre bairros adjacentes
- **Pesos ($w : E \rightarrow \mathbb{R}^+$):** Representam o custo de travessia de cada via

3.2. Algoritmos Implementados

3.2.1. Busca em Largura (BFS)

A Busca em Largura explora o grafo em níveis, visitando todos os vizinhos de um nó antes de prosseguir para o próximo nível. Complexidade: $O(V + E)$.

Aplicação: Cálculo de distâncias em número de arestas, análise de componentes conexos.

3.2.2. Busca em Profundidade (DFS)

A Busca em Profundidade explora o grafo seguindo cada ramo até sua extremidade antes de retroceder. Complexidade: $O(V + E)$.

Aplicação: Detecção de ciclos, análise de conectividade.

3.2.3. Algoritmo de Dijkstra

O algoritmo de Dijkstra determina o caminho de menor custo em grafos com pesos não-negativos. Utiliza uma fila de prioridade para selecionar o próximo nó a processar. Complexidade: $O((V + E) \log V)$.

Aplicação: Cálculo de rotas ótimas entre endereços considerando os pesos das vias.

3.2.4. Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford calcula caminhos mínimos mesmo em grafos com pesos negativos e detecta ciclos negativos. Complexidade: $O(V \cdot E)$.

Aplicação: Validação dos resultados do Dijkstra, análise robusta de caminhos.

3.3. Métricas de Rede

3.3.1. Densidade

A densidade de um grafo é definida como:

$$\rho = \frac{2|E|}{|V|(|V| - 1)}$$

Representa a proporção de conexões existentes em relação ao total possível.

3.3.2. Grau Médio

O grau médio quantifica a conectividade média da rede:

$$\langle k \rangle = \frac{1}{|V|} \sum_{v \in V} \deg(v) = \frac{2|E|}{|V|}$$

3.3.3. Ego-Network

A ego-network de um nó v é o subgrafo induzido por v e seus vizinhos imediatos, utilizada para análise local de conectividade.

4. Metodologia

4.1. Estrutura de Dados

A implementação utiliza listas de adjacência para representação eficiente do grafo. Cada nó mantém uma lista de tuplas (*vizinho, peso, metadados*).

Estrutura do Grafo

```
1 from dataclasses import dataclass
2 from typing import Dict, List, Tuple
3
4 @dataclass
5 class EdgeMeta:
6     logradouro: str | None = None
7     observacao: str | None = None
8
9 class Graph:
10     def __init__(self):
11         self._adj: Dict[str, List[Tuple[str, float, EdgeMeta]]] = {}
12
13     def add_edge(self, u: str, v: str, w: float = 1.0,
14                 meta: EdgeMeta | None = None):
15         self.add_node(u)
16         self.add_node(v)
17         self._adj[u].append((v, w, meta))
18         self._adj[v].append((u, w, meta))
19
20     def neighbors(self, u: str):
21         return self._adj.get(u, [])
```

4.2. Sistema de Cálculo de Pesos

Os pesos das arestas foram calculados utilizando a fórmula:

$$peso_{final} = (peso_{base} \times fator_{pavimentacao}) + penalidades$$

4.2.1. Componentes do Peso

Peso Base (tipo de via):

- Avenida: 1.0
- Ponte: 1.5
- Rua: 2.0
- Viaduto: 2.5

- Estrada: 3.0

Fator de Pavimentação:

- Asfalto: $\times 1.0$
- Concreto: $\times 1.0$
- Paralelepípedo: $\times 1.3$
- Escadaria: $\times 1.5$
- Sem pavimentação: $\times 2.0$
- Terra: $\times 2.0$

Penalidades:

- Ponte/Viaduto: $+0.5$
- Semáforos: $+0.3$ cada

4.3. Processamento de Dados

4.3.1. Obtenção das Interconexões

As interconexões entre bairros foram obtidas através de análise manual e consulta a múltiplas fontes:

Fontes de dados:

- **Mapas oficiais:** Mapas da Prefeitura do Recife e CONDEPE/FIDEM (Agência Estadual de Planejamento e Pesquisas de Pernambuco)
- **Google Maps:** Identificação visual de vias que conectam bairros adjacentes
- **Conhecimento local:** Validação provinda dos próprios integrantes do grupo.

Critérios para definir uma interconexão:

1. Existência de via pública (rua, avenida, ponte) conectando diretamente dois bairros
2. Via deve ser transitável por veículos ou pedestres
3. Conexão deve ser geograficamente significativa (não apenas divisa técnica)
4. Preferência para vias principais e de maior fluxo

Processo de mapeamento:

1. Identificação dos 94 bairros oficiais do Recife

2. Análise de mapas para identificar bairros adjacentes
3. Pesquisa de logradouros que fazem a ligação
4. Registro das características de cada via (tipo, pavimentação, obstáculos)
5. Validação cruzada entre diferentes fontes
6. Resultado: 244 conexões mapeadas no arquivo `adjacencias_bairros.csv`

Limitações da fonte de dados:

- Mapeamento manual pode não incluir todas as vias menores
- Mudanças urbanas recentes podem não estar refletidas
- Algumas conexões podem ser unidirecionais mas foram tratadas como bidirecionais
- Dados de pavimentação baseados em observação visual (mapas) e relatos

4.3.2. Etapa 1: Normalização dos Bairros

O dataset original continha inconsistências de nomenclatura e acentuação. Foi desenvolvido um processo de normalização que:

- Remove acentuação para chaves de busca
- Mantém a grafia original para exibição
- Resolve duplicatas e variações de nome

4.3.3. Etapa 2: Construção do Grafo

Leitura do arquivo `adjacencias_bairros.csv` contendo:

- Bairro origem e destino
- Nome do logradouro
- Observações sobre pavimentação
- Peso calculado

4.3.4. Etapa 3: Cálculo de Métricas

Aplicação dos algoritmos para extrair:

- Métricas globais (densidade, grau médio, componentes)
- Métricas por microrregião
- Ego-networks individuais
- Rankings de conectividade

4.4. Análise de Caminhos Mínimos

Foi implementado um sistema completo de análise de rotas entre endereços:

1. Normalização dos nomes dos bairros de origem e destino
2. Aplicação do algoritmo de Dijkstra
3. Reconstrução do caminho ótimo
4. Cálculo do custo total
5. Detalhamento trecho a trecho

4.5. Visualizações

O sistema gera visualizações interativas utilizando Plotly:

- **Grafo completo:** Visualização da rede completa de bairros
- **Árvore de percurso:** Destaque do caminho entre dois pontos
- **Dashboard analítico:** Métricas e estatísticas consolidadas

5. Resultados

5.1. Características da Rede

A análise revelou as seguintes características da rede de bairros do Recife:

- Número de nós (bairros): 94
- Número de arestas (conexões): 244
- Densidade da rede: $\rho \approx 0.056$
- Grau médio: $\langle k \rangle \approx 5.19$
- Componentes conexos: 1 (rede totalmente conectada)

5.2. Distribuição de Pesos

A análise dos pesos calculados das arestas mostrou:

- Peso mínimo: 1.0 (vias ideais: avenidas asfaltadas)
- Peso máximo: 6.0 (vias de maior custo)
- Peso médio: 1.71
- Mediana: 1.3

5.3. Caso de Estudo: Nova Descoberta → Boa Viagem

Foi analisado o percurso entre Nova Descoberta e Boa Viagem (região de Setúbal):

- Custo total: 10.3
- Número de bairros percorridos: 10
- Número de trechos: 9

Caminho encontrado:

1. Nova Descoberta → Córrego do Jenipapo (custo: 1.3)
2. Córrego do Jenipapo → Dois Irmãos (custo: 1.0)
3. Dois Irmãos → Caxangá (custo: 1.0)

4. Caxangá → Várzea (custo: 1.0)
5. Várzea → Curado (custo: 1.0)
6. Curado → Jardim São Paulo (custo: 1.0)
7. Jardim São Paulo → Areias (custo: 1.0)
8. Areias → Ibura (custo: 2.0)
9. Ibura → Boa Viagem (custo: 1.0)

O trecho com maior custo foi Areias → Ibura (2.0), provavelmente devido ao tipo de via ou pavimentação menos favorável.

5.4. Validação dos Algoritmos

Foi desenvolvida uma suíte completa de testes cobrindo todos os algoritmos:

5.4.1. Testes de BFS (6 casos)

- Grafo simples
- Grafo desconectado
- Estrutura em árvore
- Tratamento de nós inexistentes
- Grafos com ciclos
- Grafo completo

5.4.2. Testes de DFS (6 casos)

- Grafo simples
- Grafo desconectado
- Nós inexistentes
- Grafos com ciclos
- Ordem de visitação
- Grafo estrela

5.4.3. Testes de Dijkstra (7 casos)

- Caminho simples
- Seleção do caminho mais curto
- Nós isolados
- Pesos diferentes
- Grafo completo
- Reconstrução de caminho
- Nós inexistentes

5.4.4. Testes de Bellman-Ford (7 casos)

- Caminho simples
- Pesos positivos
- Nós isolados
- Ciclos positivos
- Grafo completo
- Nós inexistentes
- Caminho linear

Resultado: 26/26 testes passaram com sucesso, validando a correção da implementação.

5.5. Visualizações Geradas

O sistema produziu as seguintes saídas visuais:

- `out/dashboard_interativo.html`: Dashboard consolidado com todas as visualizações
- `out/arvore_percурсо.html`: Visualização interativa da árvore de percurso
- Gráficos de métricas por microrregião
- Ranking de densidade dos bairros

6. Parte 2: Dataset Maior e Comparação de Algoritmos

A segunda parte do projeto envolveu experimentos com um dataset de larga escala para avaliar o desempenho e escalabilidade dos algoritmos implementados.

6.1. Descrição do Dataset

Foi utilizado o dataset **rec-libimseti** [1, 2], obtido do Network Data Repository¹, que contém dados de um serviço de encontros online (recommender system for online dating service).

Características do dataset:

- **Número de nós ($|V|$):** 220.970
- **Número de arestas ($|E|$):** 17.359.346
- **Tipo:** Não-direcionado, ponderado
- **Grau mínimo:** 1
- **Grau máximo:** 33.389
- **Grau médio:** $\langle k \rangle \approx 157.12$

6.1.1. Distribuição de Graus

A distribuição de graus apresentou características de rede livre de escala (scale-free):

- 19.707 nós com grau 1 (nós periféricos)
- 6.903 nós com grau 2
- Concentração em graus intermediários (20-30): ~ 25.000 nós
- Pequeno número de hubs com grau muito alto (> 1000)

Esta distribuição é típica de redes reais como redes sociais, onde a maioria dos nós tem poucas conexões, mas alguns hubs centrais conectam-se a muitos outros nós.

6.2. Experimentos BFS/DFS

Foram executados BFS e DFS a partir de 3 fontes distintas selecionadas aleatoriamente.

¹<https://networkrepository.com/rec-libimseti-dir.php>

6.2.1. Resultados BFS

Fonte	Nós Alcançados	Camadas	Tempo (s)
156148	220.970	5	14.01
114556	220.970	5	13.41
164437	220.970	6	14.07

Resultados de BFS no dataset maior

Observações:

- Todas as fontes alcançaram todos os 220.970 nós (grafo conexo)
- Diâmetro efetivo: 5-6 saltos (small-world property)
- Tempo médio: ~ 13.8 segundos para $|V| + |E| \approx 17.6M$ operações

6.2.2. Resultados DFS

Fonte	Nós Visitados	Tempo (s)
156148	220.970	22.14
114556	220.970	21.19
164437	220.970	22.61

Resultados de DFS no dataset maior

Análise de Desempenho:

- DFS $\sim 60\%$ mais lento que BFS ($\sim 22s$ vs $\sim 14s$)
- Causa: Padrão de acesso à memória menos favorável ao cache
- Ambos mantêm complexidade $O(V + E)$ linear

6.3. Experimentos Dijkstra

Foram testados 5 pares origem-destino com pesos não-negativos:

Origem	Destino	Custo	Caminho	Tempo (s)
156148	83773	9.0	5 nós	21.29
31116	33639	2.0	3 nós	20.69
90280	71636	4.0	4 nós	20.04
164437	156148	19.0	6 nós	19.61
114556	182742	15.0	7 nós	20.39
Tempo médio:				20.40s

Resultados de Dijkstra no dataset maior

Análise:

- Caminhos mínimos variando de 2.0 a 19.0 (pesos unitários ou próximos)
- Tempo consistente (~ 20 s) independente do par
- Complexidade observada: $O((V + E) \log V)$
- Overhead da fila de prioridade notável em grafos densos

6.4. Experimentos Bellman-Ford

Foram testados dois casos com pesos negativos:

6.4.1. Caso 1: Pesos Negativos sem Ciclo Negativo

Grafo de teste:

- Arestas: $A \xrightarrow{4} B \xrightarrow{-2} C \xrightarrow{1} D$
- Aresta adicional: $A \xrightarrow{5} C$

Resultados:

- Origem: A
- Distâncias: $d(A) = 0, d(B) = 4, d(C) = 2, d(D) = 3$
- Ciclo negativo detectado: Não
- Tempo: 4.1×10^{-5} s (grafo pequeno)

O algoritmo corretamente encontrou que $A \rightarrow B \rightarrow C$ (custo 2) é mais barato que $A \rightarrow C$ (custo 5).

6.4.2. Caso 2: Grafo com Ciclo Negativo

Grafo de teste:

- Ciclo: $X \xrightarrow{1} Y \xrightarrow{-3} Z \xrightarrow{1} X$
- Soma de pesos no ciclo: $1 + (-3) + 1 = -1 < 0$

Resultados:

- Origem: X
- Ciclo negativo detectado: **Sim**
- Distâncias: null (indefinidas devido ao ciclo)
- Tempo: 1.9×10^{-5} s

O algoritmo detectou corretamente a presença do ciclo negativo, retornando flag de erro.

6.5. Comparação de Desempenho

Algoritmo	Complexidade	Tempo Médio	Caso de Uso
BFS	$O(V + E)$	13.8s	Caminho mais curto (não-ponderado)
DFS	$O(V + E)$	21.9s	Exploração, ciclos, conectividade
Dijkstra	$O((V + E) \log V)$	20.4s	Caminho mínimo (pesos ≥ 0)
Bellman-Ford	$O(V \cdot E)$	N/A*	Pesos negativos, detecção ciclos

Comparação de complexidade e desempenho (*não aplicável ao dataset grande devido a $V \cdot E \approx 3.8 \times 10^{15}$)

6.5.1. Análise Crítica

Quando usar cada algoritmo:

1. **BFS:** Melhor escolha para grafos não-ponderados ou quando se deseja o menor número de arestas. Mais rápido que DFS em grafos densos.
2. **DFS:** Preferível para detecção de ciclos, ordenação topológica, ou quando a estrutura em profundidade importa. Usa menos memória que BFS em grafos esparsos.
3. **Dijkstra:** Escolha padrão para caminhos mínimos em grafos com pesos não-negativos. Eficiente até centenas de milhares de nós com fila de prioridade bem implementada.
4. **Bellman-Ford:** Necessário apenas quando há pesos negativos ou quando se precisa detectar ciclos negativos. Inviável para grafos muito grandes ($|E| > 10^6$) devido à complexidade $O(V \cdot E)$.

6.5.2. Limites do Design de Pesos

O sistema de pesos implementado possui algumas limitações:

- **Pesos estáticos:** Não capturam variações temporais (tráfego, clima)
- **Heurísticas simplificadas:** Não baseadas em medições reais
- **Ausência de contexto:** Não consideram modo de transporte, acessibilidade
- **Granularidade:** Pesos discretizados podem não refletir diferenças sutis

Para aplicações críticas, seria necessário:

- Calibração com dados reais de GPS/sensoriamento
- Atualização dinâmica baseada em condições de tráfego
- Pesos dependentes de contexto (hora do dia, dia da semana)
- Modelagem multi-critério (tempo, distância, custo, segurança)

7. Discussão

7.1. Interpretação dos Resultados

A densidade relativamente baixa ($\rho \approx 0.056$) indica que a rede de bairros do Recife possui estrutura esparsa, com cada bairro conectado em média a aproximadamente 5 outros bairros. Este padrão é característico de malhas urbanas reais, onde as conexões são limitadas pela geografia e planejamento urbano.

A presença de um único componente conexo confirma que é possível transitar entre quaisquer dois bairros seguindo as conexões viárias mapeadas, característica essencial para a mobilidade urbana.

7.2. Sistema de Pesos

O sistema de pesos implementado captura adequadamente as características das vias:

- Vias principais (avenidas asfaltadas) recebem pesos menores
- Obstáculos (pontes, viadutos) aumentam o custo
- Pavimentação precária aumenta o custo de travessia

A distribuição concentrada em pesos baixos (mediana 1.0) indica que a maioria das conexões utiliza vias de boa qualidade.

7.3. Eficiência dos Algoritmos

A implementação utilizando listas de adjacência garantiu eficiência nas operações:

- BFS/DFS: $O(V + E) = O(94 + 244) \approx O(338)$
- Dijkstra: $O((V + E) \log V) \approx O(2284)$
- Bellman-Ford: $O(V \cdot E) = O(22936)$

Todos os algoritmos executam em tempo aceitável para o tamanho da rede.

7.4. Limitações

Algumas limitações identificadas neste trabalho:

7.4.1. Limitações da Coleta de Dados

- **Mapeamento manual:** As interconexões foram identificadas manualmente através de mapas e conhecimento local, podendo haver omissões de vias secundárias
- **Dados estáticos:** O mapeamento reflete um momento específico, não capturando mudanças recentes na malha urbana
- **Cobertura incompleta:** Algumas conexões viárias menores podem não estar mapeadas (244 conexões das potencialmente milhares de vias)
- **Simplificação bidirecional:** Vias unidirecionais foram tratadas como bidirecionais para simplificação
- **Ausência de dados oficiais:** Não foram utilizados dados oficiais de conectividade viária da Prefeitura (por indisponibilidade)

7.4.2. Limitações do Sistema de Pesos

- **Heurísticas não validadas:** Os pesos são estimativas baseadas em heurísticas, não medições reais de tempo ou distância
- **Fatores não considerados:** O modelo não considera tráfego dinâmico, horários de pico, condições climáticas, ou obras
- **Pesos estáticos:** Não há atualização dinâmica dos pesos baseada em condições reais
- **Granularidade limitada:** Discretização dos pesos pode não capturar diferenças sutis entre vias similares
- **Subjetividade:** Classificação de pavimentação e obstáculos baseada em observação visual de mapas

7.4.3. Limitações do Modelo

- **Modo de transporte único:** O sistema não diferencia entre deslocamento a pé, carro, bicicleta ou transporte público
- **Ausência de multimodalidade:** Não considera combinações de modos de transporte
- **Topologia simplificada:** Cada bairro é representado como um único nó, ignorando sua extensão geográfica
- **Distâncias não métricas:** Pesos não representam distâncias reais em quilômetros, mas custos abstratos

8. Código-Fonte

O código-fonte completo encontra-se disponível em repositório GitHub:

<https://github.com/MiguelBecker/Projeto-Teoria-Grafos>

8.1. Módulo Principal: Graph

Implementação da estrutura de grafo

```
1  from dataclasses import dataclass
2  from typing import Dict, List, Tuple
3
4  @dataclass
5  class EdgeMeta:
6      logradouro: str | None = None
7      observacao: str | None = None
8
9  class Graph:
10     def __init__(self):
11         self._adj: Dict[str, List[Tuple[str, float, EdgeMeta]]] = {}
12
13     def add_node(self, u: str):
14         if u not in self._adj:
15             self._adj[u] = []
16
17     def add_edge(self, u: str, v: str, w: float = 1.0,
18                 meta: EdgeMeta | None = None):
19         if meta is None:
20             meta = EdgeMeta()
21         self.add_node(u)
22         self.add_node(v)
23         self._adj[u].append((v, w, meta))
24         self._adj[v].append((u, w, meta))
25
26     def neighbors(self, u: str):
27         return self._adj.get(u, [])
28
29     def nodes(self):
30         return list(self._adj.keys())
31
32     def degree(self, u: str) -> int:
33         return len(self._adj.get(u, []))
34
35     def order(self) -> int:
36         return len(self._adj)
37
38     def size(self) -> int:
39         return sum(len(v) for v in self._adj.values()) // 2
```

8.2. Algoritmos de Busca (BFS e DFS)

Implementação de BFS e DFS

```
1  from collections import deque
2  from typing import Dict
3
4  def bfs(grafo: Graph, origem: str) -> Dict[str, int]:
5      """
6          Busca em largura (BFS) a partir de um no origem.
7          Retorna um dicionario com as distancias de cada no ate a origem.
8      """
9      if origem not in grafo.nodes():
10         return {}
11
12      distancias = {origem: 0}
13      fila = deque([origem])
14
15      while fila:
16          atual = fila.popleft()
17          dist_atual = distancias[atual]
18
19          for vizinho, _, _ in grafo.neighbors(atual):
20              if vizinho not in distancias:
21                  distancias[vizinho] = dist_atual + 1
22                  fila.append(vizinho)
23
24      return distancias
25
26  def dfs(grafo: Graph, origem: str) -> Dict[str, int]:
27      """
28          Busca em profundidade (DFS) a partir de um no origem.
29          Retorna um dicionario com a ordem de visitacao.
30      """
31      if origem not in grafo.nodes():
32          return {}
33
34      visitados = {}
35      pilha = [origem]
36      ordem = 0
37
38      while pilha:
39          no = pilha.pop()
40
41          if no in visitados:
42              continue
43
44          visitados[no] = ordem
45          ordem += 1
46
47          # Adiciona vizinhos a pilha
48          vizinhos = [v for v, _, _ in grafo.neighbors(no)]
49          for vizinho in reversed(vizinhos):
50              if vizinho not in visitados:
51                  pilha.append(vizinho)
52
53  return visitados
```

8.3. Algoritmo de Dijkstra

Implementação do algoritmo de Dijkstra

```
1 import heapq
2 from typing import Dict, Tuple
3
4 def dijkstra(grafo: Graph, origem: str) -> Tuple[Dict[str, float], Dict[str, str]]:
5     """
6         Algoritmo de Dijkstra para caminho mínimo.
7
8     Retorna:
9         distancias: dicionario {no: distancia_minima}
10        predecessores: dicionario {no: no_anterior_no_caminho}
11    """
12    distancias = {origem: 0.0}
13    predecessores = {origem: None}
14    heap = [(0.0, origem)]
15
16    while heap:
17        dist_atual, u = heapq.heappop(heap)
18
19        if dist_atual > distancias.get(u, float('inf')):
20            continue
21
22        for v, peso, _ in grafo.neighbors(u):
23            nova_dist = dist_atual + peso
24
25            if nova_dist < distancias.get(v, float('inf')):
26                distancias[v] = nova_dist
27                predecessores[v] = u
28                heapq.heappush(heap, (nova_dist, v))
29
30    return distancias, predecessores
31
32 def reconstruir_caminho(predecessores: Dict[str, str],
33                         destino: str) -> List[str]:
34     """Reconstroi o caminho a partir dos predecessores."""
35     if destino not in predecessores:
36         return []
37
38     caminho = []
39     atual = destino
40
41     while atual is not None:
42         caminho.append(atual)
43         atual = predecessores[atual]
44
45     return list(reversed(caminho))
```

8.4. Algoritmo de Bellman-Ford

Implementação do algoritmo de Bellman-Ford

```

1 def bellman_ford(grafo: Graph, origem: str) -> Tuple[Dict[str, float], Dict[str, str], bool]:
2
3     """
4         Algoritmo de Bellman-Ford para caminho minimo (aceita pesos
5             negativos).
6         Retorna:
7             - Dicionario de distancias minimas
8             - Dicionario de predecessores
9             - Bool indicando se ha ciclo negativo
10    """
11    if origem not in grafo.nodes():
12        return {}, {}, False
13
14    distancias = {no: float('inf') for no in grafo.nodes()}
15    distancias[origem] = 0.0
16    predecessores = {origem: None}
17
18    nos = grafo.nodes()
19    arestas = grafo.edges()
20
21    # Relaxamento de arestas (V-1 iteracoes)
22    for _ in range(len(nos) - 1):
23        for u, v, peso, _ in arestas:
24            if distancias[u] + peso < distancias[v]:
25                distancias[v] = distancias[u] + peso
26                predecessores[v] = u
27            # Grafo nao-direcionado: relaxa ambas direcoes
28            if distancias[v] + peso < distancias[u]:
29                distancias[u] = distancias[v] + peso
30                predecessores[u] = v
31
32    # Detecta ciclo negativo
33    tem_ciclo_negativo = False
34    for u, v, peso, _ in arestas:
35        if distancias[u] + peso < distancias[v]:
36            tem_ciclo_negativo = True
37            break
38        if distancias[v] + peso < distancias[u]:
39            tem_ciclo_negativo = True
40            break
41
42    return distancias, predecessores, tem_ciclo_negativo

```

8.5. Cálculo de Pesos

Sistema de cálculo de pesos das arestas

```

1 def calcular_peso(logradouro: str, observacao: str) -> float:
2     """
3         Calcula o peso de uma aresta baseado em caracteristicas da via.
4
5         Formula: peso_final = (peso_base * fator_pav) + penalidades
6     """
7
8     # Peso base por tipo de via
9     peso_base = 2.0 # rua padrao

```

```

9     if 'avenida' in logradouro.lower():
10        peso_base = 1.0
11    elif 'ponte' in logradouro.lower():
12        peso_base = 1.5
13    elif 'viaduto' in logradouro.lower():
14        peso_base = 2.5
15    elif 'estrada' in logradouro.lower():
16        peso_base = 3.0
17
18    # Fator de pavimentacao
19    fator_pav = 1.0
20    obs_lower = observacao.lower()
21    if 'concreto' in obs_lower:
22        fator_pav = 1.0
23    elif 'paralelepipedo' in obs_lower:
24        fator_pav = 1.3
25    elif 'escadaria' in obs_lower:
26        fator_pav = 1.5
27    elif 'sem pav' in obs_lower or 'terra' in obs_lower:
28        fator_pav = 2.0
29
30    # Penalidades
31    penalidade = 0.0
32    if 'ponte' in logradouro.lower() or 'viaduto' in logradouro.lower():
33        penalidade += 0.5
34    if 'semaforo' in obs_lower or 'sinal' in obs_lower:
35        penalidade += 0.3
36
37    return peso_base * fator_pav + penalidade

```

9. Conclusão

Para concluir, este projeto demonstrou a aplicabilidade da Teoria de Grafos para modelagem e análise de redes em diferentes escalas, desde redes urbanas locais até grandes datasets com milhões de arestas.

9.1. Parte 1: Rede de Bairros do Recife

A análise da rede de 94 bairros validou a eficácia dos algoritmos implementados:

- O sistema de pesos baseado em características reais das vias permitiu encontrar rotas otimizadas
- A validação através de 26 testes unitários garantiu a correção da implementação
- As visualizações interativas facilitaram a exploração dos dados
- O caso de estudo Nova Descoberta → Boa Viagem demonstrou aplicabilidade prática

9.2. Parte 2: Dataset de Larga Escala

Os experimentos com o dataset de 220.970 nós e 17.359.346 arestas revelaram:

- **BFS:** Melhor desempenho ($\sim 14s$) para exploração de grafos grandes
- **DFS:** Comportamento linear mantido, porém 60% mais lento que BFS
- **Dijkstra:** Escalou adequadamente para $> 200k$ nós ($\sim 20s$ por consulta)
- **Bellman-Ford:** Viável apenas para grafos pequenos devido à complexidade $O(V \cdot E)$

9.3. Lições Aprendidas

A comparação empírica confirmou as previsões teóricas:

- BFS é preferível a DFS para grafos densos (melhor localidade de cache)
- Dijkstra é adequado para grafos com centenas de milhares de nós
- Bellman-Ford deve ser reservado apenas para casos com pesos negativos
- Estruturas de dados (listas de adjacência, heaps) impactam significativamente o desempenho

Referências Bibliográficas

- [1] Ryan A. Rossi and Nesreen K. Ahmed. *The Network Data Repository with Interactive Graph Analytics and Visualization*. In AAAI, 2015. <https://networkrepository.com>
- [2] Lukas Brozovsky and Vaclav Petricek. *Recommender system for online dating service*. arXiv preprint cs/0703042, 2007.