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

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 ($w : E \to \mathbb{R}^+$):** 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

```python
from dataclasses import dataclass
from typing import Dict, List, Tuple

@dataclass
class EdgeMeta:
    logradouro: str | None = None
    observacao: str | None = None

class Graph:
    def __init__(self):
        self._adj: Dict[str, List[Tuple[str, float, EdgeMeta]]] = {}

    def add_edge(self, u: str, v: str, w: float = 1.0,
                 meta: EdgeMeta | None = None):
        self.add_node(u)
        self.add_node(v)
        self._adj[u].append((v, w, meta))
        self._adj[v].append((u, w, meta))

    def neighbors(self, u: str):
        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
- Paralelepípedo: $\times$ 1.3
- Escadaria: $\times$ 1.5
- Sem pavimentação: $\times$ 2.0

Penalidades:

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

4.3. Processamento de Dados

4.3.1. 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

4.3.2. Etapa 2: Construção do Grafo

Leitura do arquivo `adjacencias_bairros.csv` contendo:

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

4.3.3. 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:** 1.0

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_percurso.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 Repository[1], 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): $\sim$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 ~60% mais lento que BFS (~22s vs ~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 ($\sim$20s) 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 $A \to B \to C$ (custo 2) é mais barato que $A \to C$ (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:

- Os pesos são estimativas baseadas em heurísticas, não medições reais de tempo ou distância
- O modelo não considera tráfego dinâmico ou horários de pico
- Algumas conexões viárias podem não estar mapeadas
- O sistema atual não suporta múltiplos modos de transporte

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

```python
from dataclasses import dataclass
from typing import Dict, List, Tuple

@dataclass
class EdgeMeta:
    logradouro: str | None = None
    observacao: str | None = None

class Graph:
    def __init__(self):
        self._adj: Dict[str, List[Tuple[str, float, EdgeMeta]]] = {}

    def add_node(self, u: str):
        if u not in self._adj:
            self._adj[u] = []

    def add_edge(self, u: str, v: str, w: float = 1.0,
                 meta: EdgeMeta | None = None):
        if meta is None:
            meta = EdgeMeta()
        self.add_node(u)
        self.add_node(v)
        self._adj[u].append((v, w, meta))
        self._adj[v].append((u, w, meta))

    def neighbors(self, u: str):
        return self._adj.get(u, [])

    def nodes(self):
        return list(self._adj.keys())

    def degree(self, u: str) -> int:
        return len(self._adj.get(u, []))

    def order(self) -> int:
        return len(self._adj)

    def size(self) -> int:
        return sum(len(v) for v in self._adj.values()) // 2
```

8.2. Algoritmos de Busca (BFS e DFS)

Implementação de BFS e DFS

```python
from collections import deque
from typing import Dict

def bfs(grafo: Graph, origem: str) -> Dict[str, int]:
    """
    Busca em largura (BFS) a partir de um no origem.
    Retorna um dicionario com as distancias de cada no ate a origem.
    """
    if origem not in grafo.nodes():
        return {}

    distancias = {origem: 0}
    fila = deque([origem])

    while fila:
        atual = fila.popleft()
        dist_atual = distancias[atual]

        for vizinho, _, _ in grafo.neighbors(atual):
            if vizinho not in distancias:
                distancias[vizinho] = dist_atual + 1
                fila.append(vizinho)

    return distancias

def dfs(grafo: Graph, origem: str) -> Dict[str, int]:
    """
    Busca em profundidade (DFS) a partir de um no origem.
    Retorna um dicionario com a ordem de visitacao.
    """
    if origem not in grafo.nodes():
        return {}

    visitados = {}
    pilha = [origem]
    ordem = 0

    while pilha:
        no = pilha.pop()

        if no in visitados:
            continue

        visitados[no] = ordem
        ordem += 1

        # Adiciona vizinhos a pilha
        vizinhos = [v for v, _, _ in grafo.neighbors(no)]
        for vizinho in reversed(vizinhos):
            if vizinho not in visitados:
                pilha.append(vizinho)

    return visitados
```

8.3. Algoritmo de Dijkstra

Implementação do algoritmo de Dijkstra

```
import heapq
from typing import Dict, Tuple

def dijkstra(grafo: Graph, origem: str) -> Tuple[Dict[str, float], Dict[
    str, str]]:
    """
    Algoritmo de Dijkstra para caminho minimo.

    Retorna:
        distancias: dicionario {no: distancia_minima}
        predecessores: dicionario {no: no_anterior_no_caminho}
    """
    distancias = {origem: 0.0}
    predecessores = {origem: None}
    heap = [(0.0, origem)]

    while heap:
        dist_atual, u = heapq.heappop(heap)

        if dist_atual > distancias.get(u, float('inf')):
            continue

        for v, peso, _ in grafo.neighbors(u):
            nova_dist = dist_atual + peso

            if nova_dist < distancias.get(v, float('inf')):
                distancias[v] = nova_dist
                predecessores[v] = u
                heapq.heappush(heap, (nova_dist, v))

    return distancias, predecessores

def reconstruir_caminho(predecessores: Dict[str, str],
                        destino: str) -> List[str]:
    """Reconstroi o caminho a partir dos predecessores."""
    if destino not in predecessores:
        return []

    caminho = []
    atual = destino

    while atual is not None:
        caminho.append(atual)
        atual = predecessores[atual]

    return list(reversed(caminho))
```

8.4. Algoritmo de Bellman-Ford

Implementação do algoritmo de Bellman-Ford

```python
def bellman_ford(grafo: Graph, origem: str) -> Tuple[Dict[str, float],
                                                        Dict[str, str],
                                                        bool]:
    """
    Algoritmo de Bellman-Ford para caminho minimo (aceita pesos
       negativos).
    Retorna:
        - Dicionario de distancias minimas
        - Dicionario de predecessores
        - Bool indicando se ha ciclo negativo
    """
    if origem not in grafo.nodes():
        return {}, {}, False

    distancias = {no: float('inf') for no in grafo.nodes()}
    distancias[origem] = 0.0
    predecessores = {origem: None}

    nos = grafo.nodes()
    arestas = grafo.edges()

    # Relaxamento de arestas (V-1 iteracoes)
    for _ in range(len(nos) - 1):
        for u, v, peso, _ in arestas:
            if distancias[u] + peso < distancias[v]:
                distancias[v] = distancias[u] + peso
                predecessores[v] = u
            # Grafo nao-direcionado: relaxa ambas direcoes
            if distancias[v] + peso < distancias[u]:
                distancias[u] = distancias[v] + peso
                predecessores[u] = v

    # Detecta ciclo negativo
    tem_ciclo_negativo = False
    for u, v, peso, _ in arestas:
        if distancias[u] + peso < distancias[v]:
            tem_ciclo_negativo = True
            break
        if distancias[v] + peso < distancias[u]:
            tem_ciclo_negativo = True
            break

    return distancias, predecessores, tem_ciclo_negativo
```

8.5. Cálculo de Pesos

Sistema de cálculo de pesos das arestas

```python
def calcular_peso(logradouro: str, observacao: str) -> float:
    """
    Calcula o peso de uma aresta baseado em caracteristicas da via.

    Formula: peso_final = (peso_base * fator_pav) + penalidades
    """
    # Peso base por tipo de via
    peso_base = 2.0  # rua padrao
```

```
    if 'avenida' in logradouro.lower():
        peso_base = 1.0
    elif 'ponte' in logradouro.lower():
        peso_base = 1.5
    elif 'viaduto' in logradouro.lower():
        peso_base = 2.5
    elif 'estrada' in logradouro.lower():
        peso_base = 3.0

    # Fator de pavimentacao
    fator_pav = 1.0
    obs_lower = observacao.lower()
    if 'paralelepipedo' in obs_lower:
        fator_pav = 1.3
    elif 'escadaria' in obs_lower:
        fator_pav = 1.5
    elif 'sem pav' in obs_lower:
        fator_pav = 2.0

    # Penalidades
    penalidade = 0.0
    if 'ponte' in logradouro.lower() or 'viaduto' in logradouro.lower():
        penalidade += 0.5
    if 'semaforo' in obs_lower or 'sinal' in obs_lower:
        penalidade += 0.3

    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 (∼14s) para exploração de grafos grandes
- **DFS:** Comportamento linear mantido, porém 60% mais lento que BFS
- **Dijkstra:** Escalou adequadamente para $> 200k$ nós (∼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.