Manual Técnico - Projeto de Grafos: Análise da Cidade do Recife

Disciplina: Grafos

Grupo: Brandon de Oliveira Hunt, Lucas Rosati Cavalcanti Pereira, Luis Eduardo Vieira Melo, Ronaldo Tavares Souto Maior

Instituição: CESAR School

Período: 2025.2

Sumário

1. Sumário Executivo
2. Arquitetura e Estrutura
3. Parte 1 - Análise dos Bairros do Recife
4. Parte 2 - Comparação de Algoritmos
5. Implementação Técnica Detalhada
6. Guia de Uso e Visualizações
7. Fontes e Metodologia
8. Conclusões e Aprendizados

Sumário Executivo

Visão Geral

Este projeto implementa e analisa algoritmos de grafos aplicados a dados reais da cidade do Recife, dividido em duas partes complementares:

- **Parte 1**: Análise multidimensional dos 94 bairros do Recife através de múltiplas perspectivas (transporte, saúde, educação, acidentes)
- **Parte 2**: Comparação de desempenho de 4 algoritmos clássicos de grafos (BFS, DFS, Dijkstra, Bellman-Ford)

Características Técnicas

- **Linguagem:** Python 3.9+
- **Tipo de Grafo:** Não-direcionado, com arestas paralelas e pesos reais
- **Escala Parte 1:** 94 vértices (bairros), 944 arestas (vias)
- **Escala Parte 2:** 503 vértices, 4756 arestas direcionadas

---

Arquitetura e Estrutura

Estrutura de Diretórios

```
ProjetoGrafos/
├── README.md                    # Instruções de instalação e execução
├── MANUAL_TECNICO.md            # Este arquivo (documentação complet
a)
├── PARTE2_ANALISE_CRITICA.md    # Análise crítica da Parte 2
├── requirements.txt             # Dependências Python
│
├── data/                        # 41MB de dados geográficos
│   ├── adjacencias_bairros.csv  # 49KB - DADOS PRINCIPAIS DO GRAFO
│   ├── bairros.geojson          # 2.1MB - Geometrias dos bairros
│   ├── bairros_unique.csv       # 1.5KB - Bairros normalizados
│   ├── facequadra.csv           # 23MB - Dados de quadras
│   ├── distancias_vias.csv      # 47KB - Distâncias calculadas
│   └── dataset_parte2/
│       └── dataset_parte2_modificado.csv  # Dataset para Parte 2
│
├── out/
│   ├── parte1/                  # 11 arquivos (476KB)
```

```
│   │   ├── grafo_interativo.html      # Visualização principal
│   │   ├── grafo_acidentes.html       # Mapa de calor
│   │   ├── grafo_saude_ocde.html      # Índice de saúde
│   │   ├── grafo_transporte.html      # Cobertura de transporte
│   │   ├── grafo_educacao.html        # Índice educacional
│   │   ├── grafo_educacao_quadrantes.html
│   │   ├── recife_global.json         # Métricas globais
│   │   ├── microrregioes.json         # Métricas regionais
│   │   ├── ego_bairro.csv             # Densidade ego
│   │   ├── distancias_enderecos.csv   # Caminhos calculados
│   │   └── percurso_nova_descoberta_setubal.json
│   │
│   └── parte2/                        # 2 arquivos
│       ├── parte2_report.json         # Relatório comparativo
│       └── amostra_grafo.html         # Visualização interativa do grafo
│
├── src/
│   ├── cli.py                         # Interface de linha de comando
│   ├── solve.py                       # Orquestrador da Parte 2
│   ├── viz.py                         # Visualização HTML interativa (sem NetworkX)
│   │
│   ├── edges/                         # Pipeline de construção de arestas
│   │   ├── adjacencias.py             # Detecção de adjacências Queen
│   │   ├── vias_conectam.py           # Identificação de vias comuns
│   │   └── distancias.py              # Cálculo de distâncias geométricas
│   │
│   └── graphs/                        # Core do sistema de grafos
│       ├── graph.py                   # Classe principal Grafo
│       ├── algorithms.py              # BFS, DFS, Dijkstra, Bellman-Ford
│       ├── metricas.py                # Cálculo de métricas
│       ├── io.py                      # Normalização de dados
│       ├── calcular_distancias_enderecos.py
│       ├── visualizar_grafo_completo.py
│       └── visualizar_percurso.py
│
├── tests/                             # Suite de testes
```

```
│   ├── test_bfs.py
│   ├── test_dfs.py
│   ├── test_dijkstra.py
│   └── test_bellman_ford.py
│
├── Visualizacao_Acidentes/
├── Visualizaçao_Saude/
├── Visualizacao_Transporte/
└── grafico_educacao_recife/
```

Tecnologias Utilizadas

- **Python 3.9+**
- **Pandas** - Manipulação de dados
- **HTML5 Canvas** - Visualizações interativas (Partes 1 e 2)
- **JavaScript** - Frontend das visualizações (Dijkstra implementado também em JS)
- **Matplotlib** - Gráficos e análises auxiliares
- **Pytest** - Testes unitários

Importante: Todos os algoritmos (BFS, DFS, Dijkstra, Bellman-Ford) e visualizações têm implementação própria em Python. Nenhuma biblioteca externa de grafos (como NetworkX) é usada nos algoritmos ou visualizações.

Parte 1 - Análise dos Bairros do Recife

Dataset

O grafo dos bairros do Recife possui:

- **Vértices**: 94 bairros
- **Arestas**: 944 vias que conectam os bairros (233 conexões únicas)
- **Tipo**: Grafo não-direcionado com arestas paralelas
- **Pesos**: Distâncias em metros calculadas a partir de dados geográficos reais

- **Microrregiões**: 18 microrregiões administrativas

Fontes de Dados

1. **bairros_recife.xlsx** (Fonte Original)

- **Formato:** Excel com 6 colunas (microrregiões 1.1 a 6.3)
- **Conteúdo:** Lista não normalizada de bairros por microrregião
- **Problema:** Nomes inconsistentes, abreviações, bairros duplicados

2. **bairros.geojson** (2.1MB)

- **Tipo:** GeoJSON com geometrias POLYGON e MULTIPOLYGON
- **Sistema de coordenadas:** Lat/Long (graus decimais)
- **Uso:** Detecção de adjacências por contiguidade geométrica

3. **adjacencias_bairros.csv** (ARQUIVO CENTRAL - 945 linhas)

- **Estrutura:**

```
bairro_origem,bairro_destino,logradouro,peso
Alto José do Pinho,Mangabeira,Rua Aripibu,29.85
```

- **Característica crítica:** CADA LINHA = UMA ARESTA DO GRAFO
- **Total de arestas:** 944 (excluindo header)
- **Pesos:** Distâncias reais em metros (range: 29.85m a ~3000m)

Fórmula de Cálculo de Pesos

Os pesos das arestas representam **distâncias euclidianas em metros** calculadas através de:

1. **Extração de coordenadas** do GeoJSON de trechos de vias
2. **Point-in-Polygon** para verificar se trechos conectam dois bairros
3. **Cálculo de distância euclidiana:** $\sqrt{((x_2-x_1)^2 + (y_2-y_1)^2)}$
4. **Agregação** de distâncias de múltiplos trechos da mesma via

Código-fonte: `src/edges/distancias.py`

Principais Resultados

Acidentes de Trânsito (2024)

- **Total**: 5.354 acidentes registrados
- **Área crítica**: Boa Viagem (549 acidentes, 88% com vítima)
- **Top 3**: Boa Viagem, Imbiribeira (305), Santo Amaro (296)
- **Fonte**: Dados de acidentes de 2024

Índice de Saúde (Médicos por 1000 hab)

- **Média geral**: 1.8 médicos/1000 habitantes
- **Padrão OCDE**: 3.7 médicos/1000 habitantes
- **Situação crítica**: 50 bairros (53%)
- **Excelente**: 14 bairros (15%)

Transporte Público

- **Cobertura**: 99% (93 de 94 bairros)
- **237 linhas de ônibus**
- **7 estações de metrô**
- **1 linha de VLT**

Educação

- **Situação crítica**: 55 bairros (58%)
- **Paradoxo**: 53 bairros têm escolas mas alto analfabetismo
- **Excelente**: apenas 8 bairros (9%)

Métricas Calculadas

1. **Métricas Globais (** `recife_global.json` **)**

- **Ordem:** 94 vértices
- **Tamanho:** 233 arestas únicas
- **Densidade:** 0.053306
- **Fórmula da densidade:** D = 2|E| / (|V|(|V|-1)) para grafos não-direcionados

2. **Métricas por Bairro (** `ego_bairro.csv` **)**

- Grau de cada bairro
- Densidade ego (conectividade local)
- Microrregião

3. **Métricas por Microrregião (** `microrregioes.json` **)**

- 18 microrregiões analisadas
- Densidade e ordem de cada microrregião

Visualizações Interativas

O projeto gera 6 visualizações HTML totalmente interativas:

1. **Grafo Principal Interativo (** `grafo_interativo.html` **)**

- Visualização completa dos 94 bairros
- Todas as 944 vias renderizadas com curvas paralelas
- Cálculo de caminhos mínimos (Dijkstra) em tempo real no navegador
- Busca de bairros
- Destaque de vizinhos ao clicar
- Cores por microrregião
- Zoom e pan

Técnica: HTML5 Canvas com JavaScript. Dijkstra implementado **também em JavaScript** para cálculo frontend sem necessidade de backend.

2-6. **Visualizações Temáticas**

- `grafo_acidentes.html` - Mapa de calor de acidentes
- `grafo_saude_ocde.html` - Índice de saúde por bairro
- `grafo_transporte.html` - Cobertura de transporte público
- `grafo_educacao.html` - Índice educacional
- `grafo_educacao_quadrantes.html` - Análise educacional por quadrantes

Todas as visualizações são **geradas por código Python próprio** (`visualizar_grafo_*.py`), não por bibliotecas externas. O código Python gera o HTML completo com CSS e JavaScript inline.

Caminho Calculado

O projeto calcula 1 caminho específico usando Dijkstra (`distancias_enderecos.csv`):

1. **Nova Descoberta → Boa Viagem (Setúbal)**: 2682.05m (12 bairros)

Principais Insights

1. **Desigualdade espacial evidente**:
   - Zonas Sul/Centro: melhores indicadores
   - Zonas Norte/Oeste: situação crítica
2. **Paradoxo educacional**:
   - 56% dos bairros têm escolas mas alto analfabetismo
   - Indica problemas de qualidade, não acesso
3. **Transporte é universal**:
   - 99% de cobertura
   - Qualidade e frequência variam significativamente
4. **Boa Viagem concentra problemas**:
   - Líder em acidentes (549)
   - Alta densidade populacional
   - Muita movimentação

Parte 2 - Comparação de Algoritmos

Dataset de Teste

- **503 vértices** (estações)
- **4756 arestas** direcionadas
- **Pesos**: -30 a 100
- **Distribuição**: 99.9% positivas (4750 arestas), 0.1% negativas (6 arestas controladas)
- **Ciclo negativo**: 1 ciclo artificial de 3 vértices (estacao_999 ↔ estacao_998 ↔ estacao_997)

Algoritmos Testados

Algoritmo	Requisito	Status	Testes
BFS	≥ 3 origens	Cumprido	3 origens
DFS	≥ 3 origens	Cumprido	3 origens
Dijkstra	≥ 5 pares (pesos ≥ 0)	Cumprido	5 pares
Bellman-Ford	1 caso sem ciclo + 1 com ciclo	Cumprido	2 casos

Resultados de Desempenho

Algoritmo	Tempo Médio	Tempo Total	Complexidade	Uso Prático
BFS	0.50ms	1.51ms (3x)	O(V + E)	Menor nº de saltos
DFS	7.60ms	22.80ms (3x)	O(V + E)	Detecção de ciclos
Dijkstra	1.44ms	7.19ms (5x)	$O(V^2)$	Caminho mínimo (90% dos casos)
Bellman-Ford	142.61ms	285.22ms (2x)	O(V×E)	Pesos negativos, ciclos negativos

Conclusão chave: Bellman-Ford é ~99x mais lento que Dijkstra, mas é o único que detecta ciclos negativos.

Casos Testados - Bellman-Ford

Caso 1 - Peso negativo SEM ciclo:

- estacao_287 → estacao_339
- Distância: -11
- Ciclo detectado: Não
- Tempo: 8.42ms

Caso 2 - Ciclo negativo detectado:

- estacao_999 → estacao_997
- Ciclo confirmado: Sim
- Peso do ciclo: -5
- Tempo: 276.80ms

Quando Usar Cada Algoritmo

1. **BFS** - Menor número de saltos, ignora pesos
    - Redes sociais (grau de separação)
    - Redes de computadores (menor latência)
2. **DFS** - Detecção de ciclos, busca em profundidade
    - Detecção de dependências circulares
    - Resolução de labirintos
    - Topological sort
3. **Dijkstra** - Caminho mínimo com pesos não-negativos
    - GPS/Navegação (90% dos casos)
    - Roteamento de redes
    - Logística e distribuição
4. **Bellman-Ford** - Pesos negativos, detecção de ciclos negativos
    - Sistemas financeiros (arbitragem)
    - Detecção de fraudes

- Validação de sistemas

Regra de ouro: Use Dijkstra se puder, Bellman-Ford se precisar.

Implementação Técnica Detalhada

Estrutura de Dados

Classe Grafo (`src/graphs/graph.py`)

```
class Grafo:
    def __init__(self):
        self.vertices = set()     # Set[str] - Nomes dos bairros
        self.arestas = []         # List[Dict] - Lista de dicionários
```

Por que lista de dicionários e não matriz de adjacências?

1. **Grafo esparso:** 944 arestas para 94^2 = 8,836 possíveis pares
   - Densidade = 0.214 (apenas 21.4% das conexões possíveis existem)
   - Lista usa ~37KB vs matriz usaria ~70KB + complexidade
2. **Arestas paralelas:** Matriz não suporta múltiplas arestas por par
3. **Metadados ricos:** Cada aresta carrega logradouro e peso

Algoritmos Implementados

Todos os algoritmos estão em `src/graphs/algorithms.py` com **implementação própria**.

1. BFS (Busca em Largura)

```
def bfs(grafo, origem):
    """Busca em Largura - explora por níveis, ignora pesos"""
    visitados = set()
    fila = deque([(origem, 0)])
    ordem_visitacao = []
    niveis = {}
```

```
    while fila:
        vertice_atual, nivel = fila.popleft()

        if vertice_atual in visitados:
            continue

        visitados.add(vertice_atual)
        ordem_visitacao.append(vertice_atual)

        if nivel not in niveis:
            niveis[nivel] = []
        niveis[nivel].append(vertice_atual)

        for destino, _ in grafo.obter_vizinhos(vertice_atual):
            if destino not in visitados:
                fila.append((destino, nivel + 1))

    return {
        'ordem_visitacao': ordem_visitacao,
        'niveis': niveis,
        'total_visitados': len(visitados)
    }
```

Complexidade: O(V + E)

2. DFS (Busca em Profundidade)

```
def dfs(grafo, origem):
    """Busca em Profundidade - explora em profundidade, detecta ciclos"""
    visitados = set()
    ordem_visitacao = []
    ciclos_detectados = []
    pilha_recursao = set()

    def dfs_recursivo(vertice, pai=None):
        visitados.add(vertice)
```

```
        pilha_recursao.add(vertice)
        ordem_visitacao.append(vertice)

        for vizinho, _ in grafo.obter_vizinhos(vertice):
            destino = vizinho['destino'] if isinstance(vizinho, dict) else vizinho

            if destino not in visitados:
                dfs_recursivo(destino, vertice)
            elif destino in pilha_recursao and destino != pai:
                # Ciclo detectado!
                ciclos_detectados.append((vertice, destino))

        pilha_recursao.remove(vertice)

    dfs_recursivo(origem)

    return {
        'ordem_visitacao': ordem_visitacao,
        'total_visitados': len(visitados),
        'ciclos_detectados': len(ciclos_detectados) > 0,
        'ciclos': ciclos_detectados
    }
```

Complexidade: O(V + E)

3. Dijkstra

```
def dijkstra(grafo, origem, destino):
    """Dijkstra - caminho mínimo com pesos não-negativos"""
    distancias = {v: float('inf') for v in grafo.vertices}
    distancias[origem] = 0
    anteriores = {v: None for v in grafo.vertices}
    visitados = set()

    while len(visitados) < len(grafo.vertices):
        # Seleciona vértice não visitado com menor distância (O(V))
```

```
    atual = None
    menor_distancia = float('inf')

    for v in grafo.vertices:
        if v not in visitados and distancias[v] < menor_distancia:
            menor_distancia = distancias[v]
            atual = v

    if atual is None or distancias[atual] == float('inf'):
        break

    visitados.add(atual)

    if atual == destino:
        break

    # Relaxamento
    for vizinho, peso in grafo.obter_vizinhos(atual):
        if vizinho not in visitados:
            nova_distancia = distancias[atual] + peso

            if nova_distancia < distancias[vizinho]:
                distancias[vizinho] = nova_distancia
                anteriores[vizinho] = atual

# Reconstruir caminho
caminho = []
atual = destino
while atual is not None:
    caminho.append(atual)
    atual = anteriores[atual]
caminho.reverse()

return {
    'distancia': distancias[destino],
    'caminho': caminho,
```

```
        'sucesso': caminho is not None
    }
```

Complexidade: O(V² + E) - sem heap

Otimização possível: Com heap, reduziria para O((V + E) log V)

4. Bellman-Ford

```
def bellman_ford(grafo, origem, destino):
    """Bellman-Ford - detecta ciclos negativos"""
    distancias = {v: float('inf') for v in grafo.vertices}
    distancias[origem] = 0
    anteriores = {v: None for v in grafo.vertices}

    # Relaxamento V-1 vezes
    for _ in range(len(grafo.vertices) - 1):
        for aresta in grafo.arestas:
            u = aresta['origem']
            v = aresta['destino']
            peso = aresta['peso']

            if distancias[u] != float('inf') and \\
               distancias[u] + peso < distancias[v]:
                distancias[v] = distancias[u] + peso
                anteriores[v] = u

    # Verifica ciclo negativo
    ciclo_negativo = False
    for aresta in grafo.arestas:
        u = aresta['origem']
        v = aresta['destino']
        peso = aresta['peso']

        if distancias[u] != float('inf') and \\
           distancias[u] + peso < distancias[v]:
```

```
            ciclo_negativo = True
            break

    if ciclo_negativo:
        return {
            'distancia': None,
            'caminho': None,
            'sucesso': False,
            'ciclo_negativo': True
        }

    # Reconstruir caminho
    caminho = []
    atual = destino
    while atual is not None:
        caminho.append(atual)
        atual = anteriores[atual]
    caminho.reverse()

    return {
        'distancia': distancias[destino],
        'caminho': caminho,
        'sucesso': True,
        'ciclo_negativo': False
    }
```

Complexidade: O(V × E)

Para este projeto: O(503 × 4756) ≈ 2.4 milhões de operações

5. Visualização Interativa (`viz.py`)

O módulo `viz.py` implementa visualização de grafos sem usar NetworkX, gerando HTML5 Canvas puro com JavaScript.

Função Principal: `gerar_visualizacao_amostra_grafo()`

```
def gerar_visualizacao_amostra_grafo(grafo, num_vertices=50, output_path
='out/parte2/amostra_grafo.html'):
    """
    Gera visualização interativa em HTML/Canvas (SEM NetworkX)

    Args:
        grafo: Grafo direcionado
        num_vertices: Número de vértices a incluir na amostra
        output_path: Caminho para salvar a visualização HTML
    """
```

Processo de geração:

1. **Amostragem**: Seleciona aleatoriamente N vértices (padrão: 50)
2. **Filtragem**: Remove vértices isolados (sem conexões na amostra)
3. **Layout circular**: Posiciona vértices em círculo usando trigonometria
4. **Geração HTML**: Cria arquivo HTML completo com CSS e JavaScript inline

Recursos implementados:

- **Canvas interativo** com zoom e pan
- **Arestas direcionadas** com setas
- **Cores por peso**: Verde (positivo) / Vermelho (negativo)
- **Tooltips dinâmicos** ao passar o mouse
- **Estatísticas em tempo real** (vértices, arestas, distribuição)
- **Controles de zoom** programáticos e por scroll
- **Tracejado diferencial** para arestas negativas

Por que não usar NetworkX?

1. **Controle total**: Layout e renderização customizados
2. **Interatividade**: Canvas permite zoom/pan fluidos
3. **Educacional**: Implementação própria demonstra compreensão

4. **Portabilidade**: HTML funciona em qualquer navegador

Complexidade: O(V + E) para geração do HTML

Guia de Uso e Visualizações

Como Abrir as Visualizações

Método 1: Abrir Todas de Uma Vez

```
cd ProjetoGrafos
open out/parte1/*.html
```

Isso abrirá **6 abas** no seu navegador com todas as visualizações.

Método 2: Abrir Individual

```
# Visualização principal (recomendado começar por aqui!)
open out/parte1/grafo_interativo.html

# Acidentes de trânsito
open out/parte1/grafo_acidentes.html

# Saúde
open out/parte1/grafo_saude_ocde.html

# Transporte público
open out/parte1/grafo_transporte.html

# Educação
open out/parte1/grafo_educacao.html

# Educação por quadrantes
open out/parte1/grafo_educacao_quadrantes.html
```

Controles Interativos

Todas as visualizações HTML têm os mesmos controles:

- **Arrastar**: Move o mapa
- **Scroll do mouse**: Zoom in/out
- **Clicar em um bairro**: Destaca conexões
- **Campo de busca**: Encontrar bairro específico
- **Dropdown de filtros**: Filtrar por microrregião
- **Passar o mouse**: Ver informações detalhadas
- **Botão Resetar**: Voltar à visualização inicial

Ver Dados da Parte 1

```
# Métricas globais
cat out/parte1/recife_global.json

# Métricas por microrregião
cat out/parte1/microrregioes.json

# Densidade ego por bairro
open out/parte1/ego_bairro.csv

# Caminhos calculados
open out/parte1/distancias_enderecos.csv

# Detalhes de um caminho específico
cat out/parte1/percurso_nova_descoberta_setubal.json
```

Ver Dados da Parte 2

```
# Relatório comparativo
cat out/parte2/parte2_report.json

# Ou formatado
```

```
python3 -m json.tool out/parte2/parte2_report.json

# Visualização interativa do grafo
open out/parte2/amostra_grafo.html
```

Controles da visualização HTML:

- **Arrastar**: Move o grafo
- **Scroll**: Zoom in/out
- **Hover**: Ver informações do vértice
- **Botões**: Resetar visualização, Zoom +/-
- **Cores**: Verde = arestas positivas, Vermelho = arestas negativas

---

Fontes e Metodologia

Fontes de Dados

1. **Dados Geográficos**: GeoJSON oficial da cidade do Recife
   - Geometrias dos bairros (POLYGON/MULTIPOLYGON)
   - Trechos de logradouros (LineString/MultiLineString)
   - Sistema de coordenadas: Lat/Long (graus decimais)
2. **Dados de Acidentes**: Registros de acidentes de trânsito de 2024
3. **Dados de Saúde**: Número de médicos por bairro
4. **Dados de Educação**: Presença de escolas e taxa de analfabetismo
5. **Dados de Transporte**: Linhas de ônibus, estações de metrô e VLT

Metodologia de Processamento

Pipeline de Dados da Parte 1:

```
1. NORMALIZAÇÃO
   bairros_recife.xlsx → [io.py] → bairros_unique.csv
```

```
  ↓
  - Remove abreviações (Sto. → Santo, Sta. → Santa)
  - Capitaliza corretamente
  - Agrupa microrregiões

2. DETECÇÃO DE ADJACÊNCIAS
  bairros.geojson + bairros_unique.csv → [adjacencias.py] → adjacencias_provisorio.csv
  ↓
  - Extrai vértices de polígonos
  - Aplica adjacência Queen (compartilha qualquer ponto)
  - 94 bairros → ~200 pares adjacentes

3. MAPEAMENTO DE VIAS
  adjacencias_provisorio.csv + facequadra.csv → [vias_conectam.py] → vias_conectam_bairros.csv
  ↓
  - Join por código de logradouro
  - Identifica vias compartilhadas

4. CÁLCULO DE DISTÂNCIAS
  vias_conectam_bairros.csv + geometrias → [distancias.py] → distancias_vias.csv
  ↓
  - Point-in-polygon para cada trecho
  - Soma comprimentos euclidianos
  - ~47KB de distâncias

5. CONSOLIDAÇÃO FINAL
  distancias_vias.csv → adjacencias_bairros.csv
  ↓
  - 944 arestas com pesos reais em metros
```

Pipeline de Análise:

```
1. CONSTRUÇÃO DO GRAFO
   adjacencias_bairros.csv → [graph.py] → Grafo

2. CÁLCULO DE MÉTRICAS
   Grafo → [metricas.py] → recife_global.json, microrregioes.json, ego_bairro.csv

3. CAMINHOS ENTRE ENDEREÇOS
   Grafo → [calcular_distancias_enderecos.py] → distancias_enderecos.csv

4. VISUALIZAÇÃO
   Grafo + métricas → [visualizar_grafo_completo.py] → grafo_interativo.html
```

Limitações

1. **Dados temporais**: Análise baseada em snapshot único (2024)
2. **Arestas paralelas**: Algoritmo Dijkstra retorna primeira via encontrada, não necessariamente a do caminho ótimo
3. **Escalabilidade**: Implementação $O(V^2)$ do Dijkstra não é ideal para grafos com >1000 vértices
4. **Normalização de nomes**: Alguns bairros podem ter inconsistências residuais

---

Conclusões e Aprendizados

Técnicos

1. **Algoritmos têm trade-offs claros**:
   - Velocidade vs Generalidade
   - Simplicidade vs Funcionalidade
2. **Dados reais são complexos**:
   - Limpeza é 70% do trabalho
   - Normalização é essencial

3. **Visualização é poderosa**:
    - Facilita identificação de padrões
    - Comunica resultados efetivamente
4. **Implementação própria ensina mais**:
    - Entendimento profundo dos algoritmos
    - Controle total sobre o comportamento
    - Código educacional e bem documentado

Práticos

1. **Planejamento é crucial**:
    - Estrutura de dados bem definida
    - Modularização do código
    - Testes desde o início
2. **Documentação é investimento**:
    - README detalhado economiza tempo
    - Comentários no código facilitam manutenção
    - Manual técnico permite replicação
3. **Iteração é natural**:
    - Dataset modificado para atender requisitos
    - Algoritmos ajustados conforme necessidade
    - Visualizações refinadas com feedback

Sobre a Cidade do Recife

1. **Desigualdade espacial evidente**:
    - Zonas Sul/Centro: melhores indicadores
    - Zonas Norte/Oeste: situação crítica
2. **Paradoxo educacional**:

- 56% dos bairros têm escolas mas alto analfabetismo
- Indica problemas de qualidade, não acesso

3. **Transporte é universal, mas**:
    - 99% de cobertura
    - Qualidade e frequência variam

Sobre Algoritmos de Grafos

1. **Não existe algoritmo universal**:
    - Cada um tem seu nicho específico
    - Trade-off entre velocidade e generalidade
2. **Regra de ouro**:
    - "Use Dijkstra se puder, Bellman-Ford se precisar"
3. **Performance vs Necessidade**:
    - BFS/DFS: análise estrutural rápida
    - Dijkstra: 95% dos casos práticos
    - Bellman-Ford: casos especiais (mas essenciais)
4. **Ciclos negativos são raros mas críticos**:
    - Indicam erros de design ou vulnerabilidades
    - Apenas Bellman-Ford detecta
    - Essencial para validação de sistemas financeiros e detecção de arbitragem

---

Grupo:

- Brandon de Oliveira Hunt
- Lucas Rosati Cavalcanti Pereira
- Luis Eduardo Vieira Melo
- Ronaldo Tavares Souto Maior

Disciplina: Grafos

Instituição: CESAR School

Período: 2025.2