FACULDADE CESAR SCHOOL

CURSO DE GRADUAÇÃO EM 2025.2

Manual do projeto de Grafos

DAVI GOMES FERREIRA RUY DE ALMEIDA, HELOÍSA TANAKA FERNANDES, ISABELA SPINELLI FERRARI DE SIQUEIRA CAMPOS ARRUDA, JOÃO PEDRO FONTES FERREIRA & MARIA LUÍSA VIEIRA ARRUDA

RECIFE 11/2025

1. Introdução

Este projeto teve como objetivo aplicar os conceitos de teoria dos grafos em diferentes contextos, unindo análise de dados reais, implementação de algoritmos e visualização de resultados. O trabalho foi dividido em duas partes principais, cada uma com desafios e finalidades complementares.

Na primeira parte, foi desenvolvido o Grafo dos Bairros do Recife, representando as conexões entre os bairros da cidade a partir de suas interligações geográficas. Essa etapa envolveu o processamento de dados, definição de pesos, cálculo de métricas estruturais e

criação de visualizações que ajudam a compreender a conectividade urbana. Além disso, foram implementados algoritmos clássicos de grafos para explorar percursos e relações entre os bairros.

Na segunda parte, o foco foi a análise e comparação de algoritmos de grafos utilizando o dataset “Ligações Rodoviárias e Hidroviárias 2016” do IBGE, que descreve conexões entre municípios brasileiros a partir do transporte público intermunicipal. A partir desses dados, foi construído um grafo ponderado e direcionado, em que os vértices representam os municípios e as arestas correspondem às ligações de transporte entre eles, com pesos definidos pelo tempo de deslocamento. Essa base permitiu investigar o comportamento dos algoritmos em uma rede real de grande escala, observando diferenças de desempenho e eficiência em contextos com pesos diversos.

De forma geral, o projeto buscou integrar teoria e prática, demonstrando como os grafos podem ser utilizados para representar e compreender sistemas reais, desde a estrutura urbana de uma cidade até as redes de transporte que interligam municípios em todo o território nacional. A comparação entre os dois contextos analisados evidenciou como diferentes algoritmos se adaptam a redes com topologias e escalas distintas, reforçando a versatilidade da teoria dos grafos na análise e solução de problemas do mundo real.

2. Estrutura do projeto

A organização do diretório projeto-grafos/ é a seguinte:

```
projeto-grafos/
├─ README.md
├─ requirements.txt
├─ data/
  ├─ bairros_recife.csv
  ├─ adjacencias_bairros.csv
  ├─ enderecos.csv
  └─ dataset_parte2/
    ├─ enderecos_parte2.csv
    └─ LRH2016_00_Base_Completa.csv
├─ lib/
  ├─ bindings/
        └─ utils.js
  ├─ tom-select/
        ├─tom-select.complete.min.js
        └─ tom-select.css
  └─ vis-9.1.2/
        ├─ vis-network.css
        └─ vis-network.min.js
├─ out/
  └─ parte2/
        ├─ BFS
              ├─ bfs_Foz do Iguaçu_Arroio do Sal.json
```

```
        ├─ bfs_Jataizinho_Goiorê.json
        ├─ bfs_Pouso Redondo_Xanxerê.json
        ├─ bfs_Recife_Joinville.json
        └─ bfs_São Paulo_Maragogi.json
    ├─ DFS
        ├─ dfs_Foz do Iguaçu_Arroio do Sal.json
        ├─ dfs_Jataizinho_Goiorê.json
        ├─ dfs_Pouso Redondo_Xanxerê.json
        ├─ dfs_Recife_Joinville.json
        └─ dfs_São Paulo_Maragogi.json
    ├─ distancias_enderecos_parte2_bellman_ford.csv
    ├─ distancias_enderecos_parte2.csv
    ├─ parte2_report.json
    ├─ percurso_foz_do_iguacu_arroio_do_sal.json
    ├─ percurso_jataizinho_goioere.json
    ├─ percurso_palmitos_sao_jose_do_rio_preto.json
    ├─ percurso_pouso_redondo_xanxere.json
    ├─ percurso_recife_joinville.json
    ├─ percurso_sao_carlos_julio_de_castilhos.json
    └─ percurso_sao_paulo_maragogi.json
  └─ visualizacoesPt1
    ├─ histograma_graus_freq.csv
    ├─ histograma_graus_nota.txt
    ├─ histograma_graus.png
    ├─ mapa_cores_nota.txt
    ├─ mapa_cores.html
    ├─ ranking_densidade_ego_microrregiao_nota.txt
    ├─ ranking_densidade_ego_microrregiao.csv
    ├─ ranking_densidade_ego_microrregiao.png
    ├─ subgrafo_top10_grau_nota.txt
    └─ subgrafo_top10_grau.html
└─ visualizacoesPt2
    ├─ amostra_grafo.html
    ├─ distribuicao_graus.png
    └─ heatmap_distancias.png
├─ src/
  ├─ cli.py
  ├─ solve.py
  ├─ gerar_arvore_percurso.py
  ├─ get_bairro_maior_grau.py
  ├─ get_bairro_mais_denso.py
  ├─ get_bairro-grau.py
  ├─ global_metrics_calculator.py
  ├─ graphs/
    ├─ io.py
    ├─ graph.py
    └─ algorithms.py
  ├─ viz.py
```

```
├─ tests/
  ├─ cli_parte2.py
  ├─ test_bfs.py
  ├─ test_dfs.py
  ├─ test_dijkstra.py
  └─ test_bellman_ford.py
```

Pastas principais:

- data/ – armazena todos os arquivos de entrada usados pelos algoritmos (datasets de bairros, adjacências, endereços e dados da parte 2).

- out/ – guarda os resultados gerados pelo projeto, como arquivos CSV e JSON com métricas e percursos calculados (incluindo a subpasta parte2/ para a segunda parte do trabalho).

- src/ – contém o código-fonte da aplicação: scripts principais, funções de cálculo e a implementação da estrutura de grafos e algoritmos.

- tests/ – reúne os scripts geradores dos testes de cada algoritmo.

3. Parte 1 - Grafo dos bairros do Recife

3.1. Fonte de Dados

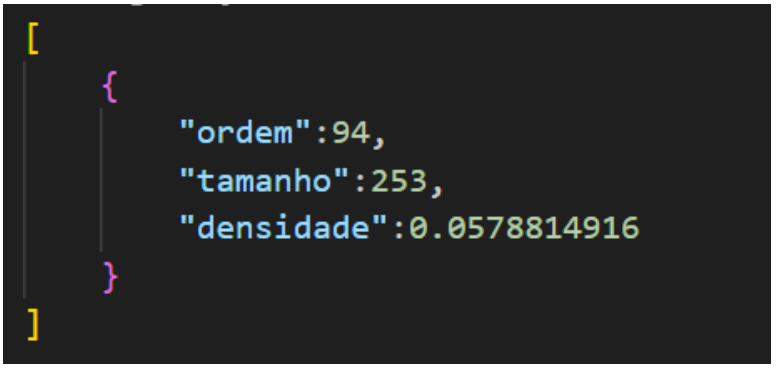
O arquivo **bairros_recife.csv** é uma tabela que contém informações sobre os bairros da cidade do Recife, organizada de forma que cada coluna representa uma microrregião, e as linhas correspondem aos bairros pertencentes a essas microrregiões. Para facilitar a análise, esse arquivo foi tratado e transformado por meio de um processo de "unpivot", onde os dados são reorganizados de modo que cada linha representa um bairro associado à sua microrregião, resultando em um formato mais simples e limpo. A transformação também incluiu a normalização dos nomes dos bairros para garantir consistência e facilitar a comparação entre eles. A partir desse processo, foi gerado o arquivo **bairros_unique.csv**, utilizando o script **io.py**, que automatiza a conversão e a padronização dos dados.

O arquivo **adjacencias_bairros.csv** foi criado para representar as adjacências entre os bairros de Recife, contendo informações sobre quais bairros são vizinhos e os pesos associados a essas adjacências. Para construir o arquivo,foram utilizados os mapas da Prefeitura do Recife, que fornecem as delimitações de cada bairro, para identificar quais bairros são diretamente vizinhos. Os pesos foram calculados com o auxílio do Google Maps, determinando as rotas mais relevantes entre os bairros e atribuindo o peso de acordo com a quilometragem das ruas conectando-os.

3.2. Métricas do Grafo

Para calcular as métricas globais do grafo, foi desenvolvido o script **global_metrics_calculator.py**, que gerou três arquivos principais: **recife_global.json**, que contém a ordem, tamanho e densidade global do grafo completo; **microrregioes.json**, que apresenta a ordem, tamanho e densidade de cada microrregião; e **ego_bairro.csv**, que reúne informações sobre cada bairro, incluindo grau, ordem, tamanho e densidade, permitindo uma análise detalhada da conectividade local.

Além das métricas globais, foram criados scripts adicionais para gerar informações mais específicas. O arquivo **graus.csv** apresenta o grau de cada bairro em relação à sua microrregião, enquanto **bairro_mais_denso.csv** lista os 10 bairros mais densos, com suas respectivas densidades. Já o arquivo **bairro_maior_grau.csv** contém os bairros com os maiores graus de conectividade, destacando os dois bairros com maior grau



Resultado de **recife_global.json**

3.3. Algoritmo

O Algoritmo de **Dijkstra** foi implementado manualmente no arquivo **algorithms.py**, utilizando apenas estruturas de dados nativas do **Python** e a estrutura de fila de prioridade fornecida pela biblioteca padrão **heapq**.

A estrutura do grafo utilizada pelo algoritmo é baseada em lista de adjacência, definida em **graph.py**, onde para cada vértice é mantida uma lista de pares (vizinho, peso). A função que executa Dijkstra é **dijkstra_path()**, que recebe um grafo, o vértice inicial e o vértice final.

A implementação segue a lógica clássica do algoritmo: mantém-se uma fila de prioridade onde cada elemento contém um par (**custo_acumulado**, nó_atual). Duas estruturas centrais controlam o avanço e o resultado do algoritmo:

- **custo_minimo**: Armazena para cada nó a melhor distância provisória encontrada até o momento.
- **no_predecessor**: Mantém a lista de predecessores que serão utilizados para reconstruir o caminho mínimo ao final.

O processo começa inserindo a origem na fila com custo 0 e registrando esse valor em **custo_minimo**. Em seguida, o loop principal retira da fila o nó com menor custo acumulado. Se o **custo_acumulado** for maior que o valor já registrado em **custo_minimo**

para o **nó_atual**, significa que um caminho melhor já havia sido encontrado e processado para aquele nó, e portanto o nó é ignorado. Se o **nó_atual** for o destino, o algoritmo é encerrado imediatamente, pois seu custo mínimo foi fixado.

Para cada vizinho do nó expandido, calcula-se um custo potencial para alcançá-lo: **novo_custo_total = custo_atual + peso_aresta**. Se o novo_custo_total for menor que o custo atualmente registrado em **custo_minimo** para o vizinho, sua distância provisória é atualizada, o predecessor correspondente é salvo em **no_predecessor** e o vizinho é enfileirado novamente na fila de prioridade.

Esse processo garante que, quando um nó é finalmente alcançado e retirado, seu menor custo já foi encontrado, refletindo a propriedade fundamental do algoritmo de Dijkstra. A construção do caminho final é feita percorrendo a cadeia de predecessores registrada em **no_predecessor**, partindo do destino até a origem, invertendo a ordem ao final (**melhor_caminho[::-1]**). Caso não haja caminho, retorna uma lista vazia [].

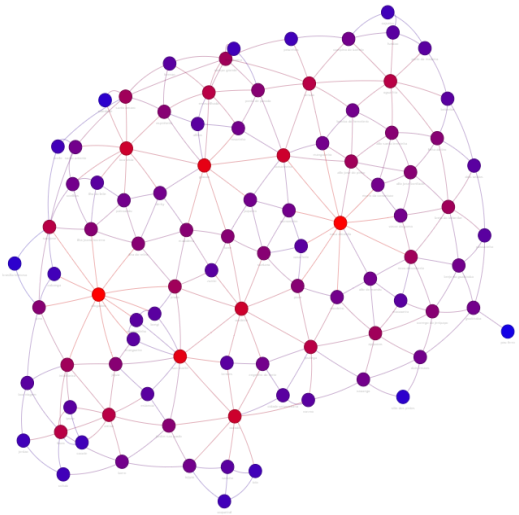
Além disso, a implementação inclui verificações importantes, como o levantamento da exceção **ValueError** caso algum dos vértices não exista no grafo ou se houver um peso negativo.

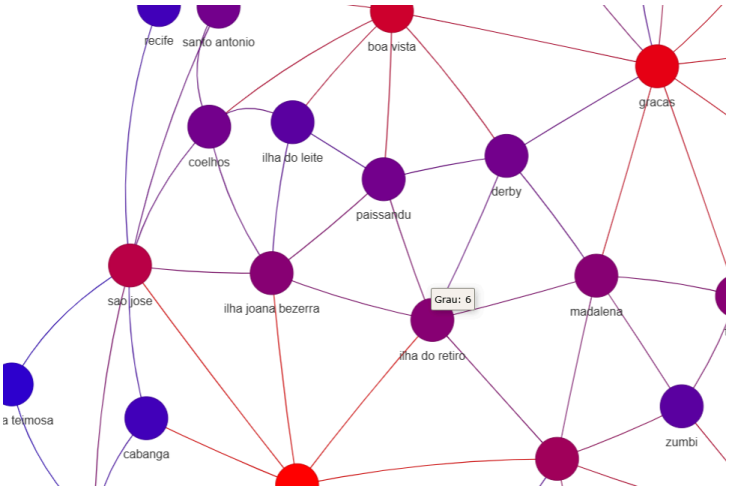
A execução do algoritmo de Dijkstra segue o fluxo definido no arquivo **src/cli.py**, que disponibiliza a flag **--calc-enderecos** para acionar o cálculo automático dos menores caminhos entre bairros. Para rodar o Dijkstra, basta executar no terminal, dentro do diretório principal do projeto, o comando **python -m src.cli --calc-enderecos**. Este comando carrega o grafo a partir de **adjacencias_bairros.csv**, processa todos os pares presentes em **enderecos.csv** e aplica o algoritmo de Dijkstra implementado em **algorithms.py** para calcular o custo e o percurso mínimo entre cada origem e destino. Ao fim da execução, são gerados automaticamente os arquivos **out/distancias_enderecos.csv**, contendo todos os resultados consolidados, além dos arquivos JSON individuais para cada par processado e o arquivo obrigatório **out/percurso_nova_descoberta_setubal.json**. Todo o cálculo do caminho e construção do percurso é feito internamente por Dijkstra, sem necessidade de parâmetros adicionais.

3.4. Visualizações Analíticas

Para as quatro visualizações, a base utilizada foi o grafo derivado de ‘data/adjcencias_bairros.csv’, grafo não direcionado, no qual cada aresta representa uma adjacência física entre bairros.

3.4.1. Mapa de Graus por Bairro (PyVis)





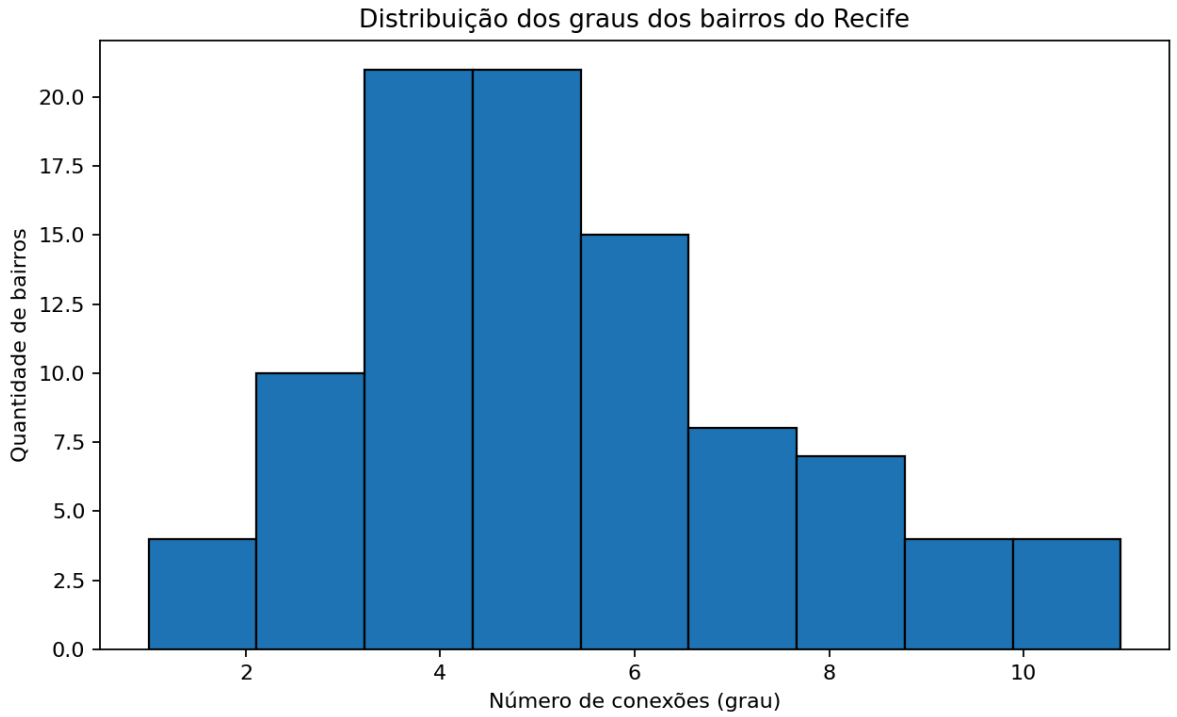
Visualização interativa, que viabiliza a exposição de todos os bairros do recife de colorindo seus nós, cada bairro, conforme seus graus (número de conexões).

- Nós = bairros
- Arestas = fronteiras físicas
- Cor = intensidade do grau
- Tamanho do nó = proporção ao grau

Insight obtido:

A partir desta visualização, que evidencia quais bairros são hubs estruturais (aqueles que fazem fronteira com muitos outros), é possível determinar quais bairros tendem a desempenhar um papel central na conectividade da cidade, funcionando como regiões de passagem e de alta interligação. Graças a essa visualização, a determinação de hubs estruturais ficou bem mais evidente.

3.4.2. Histograma dos Graus dos Bairros (Matplotlib)



O gráfico de barras exposto acima constata a distribuição da quantidade de conexões, no grafo representado por graus, para todos os bairros do Recife.

Insight obtido:

A distribuição observada não é uniforme:

- Há muitos bairros com baixa conectividade.
- Um grupo menor apresenta graus altos (mais conexões).

Isso deixa claro que a malha dos bairros de Recife trata-se de uma rede heterogênea, com poucos bairros centrais e muitos periféricos, sendo este um comportamento típico de grandes redes urbanas.

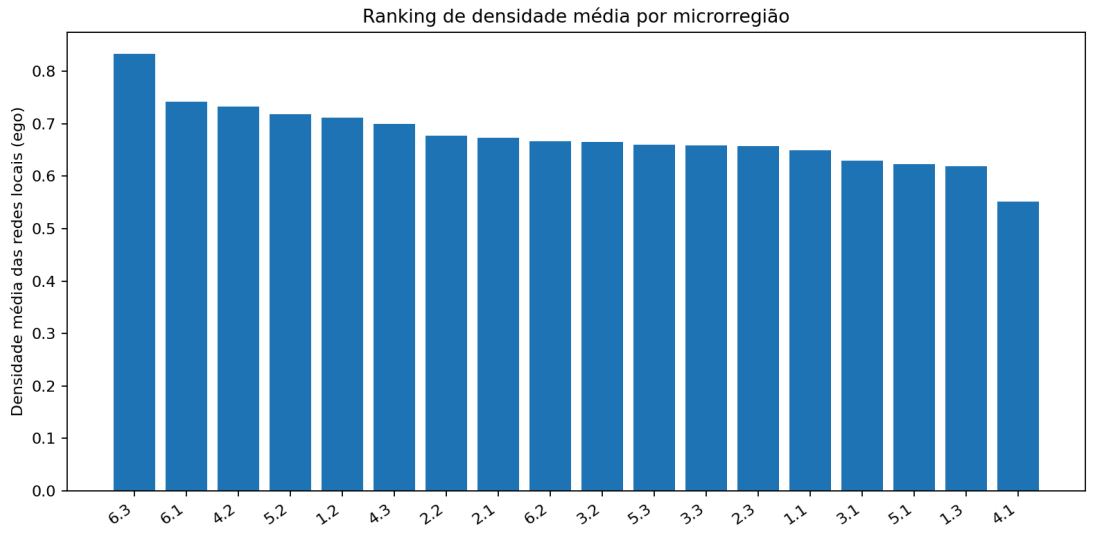
3.4.3 Subgrafo dos 10 Bairros com Maior Grau (PyVis)

Um recorte do grafo completo contendo apenas os 10 bairros mais conectados, mantendo apenas arestas entre eles.

Insight obtido:

O subgrafo mostra um núcleo urbano altamente interligado, onde os bairros mais centrais formam um bloco compacto, com maior densidade de relações. Esse "core" da rede é responsável pela maior parte da robustez estrutural do grafo. Com isso, é confirmado novamente o comportamento heterogêneo da rede urbana recifense, já que com base nas nossas análises do mapa recifense, todos esses se concentram geograficamente, excluindo bairros periféricos dessa representação.

3.4.4. Ranking de Densidade das Ego-Networks por Microrregião (PNG/CSV)



Para cada microrregião, foi construída a *ego-network* de cada bairro (nó + vizinhos diretos). Em seguida calculou-se sua densidade local (quantas conexões existem entre os vizinhos). O gráfico apresenta, para cada microrregião, o bairro com maior densidade.

Insight obtido:

A densidade revela coerência interna:

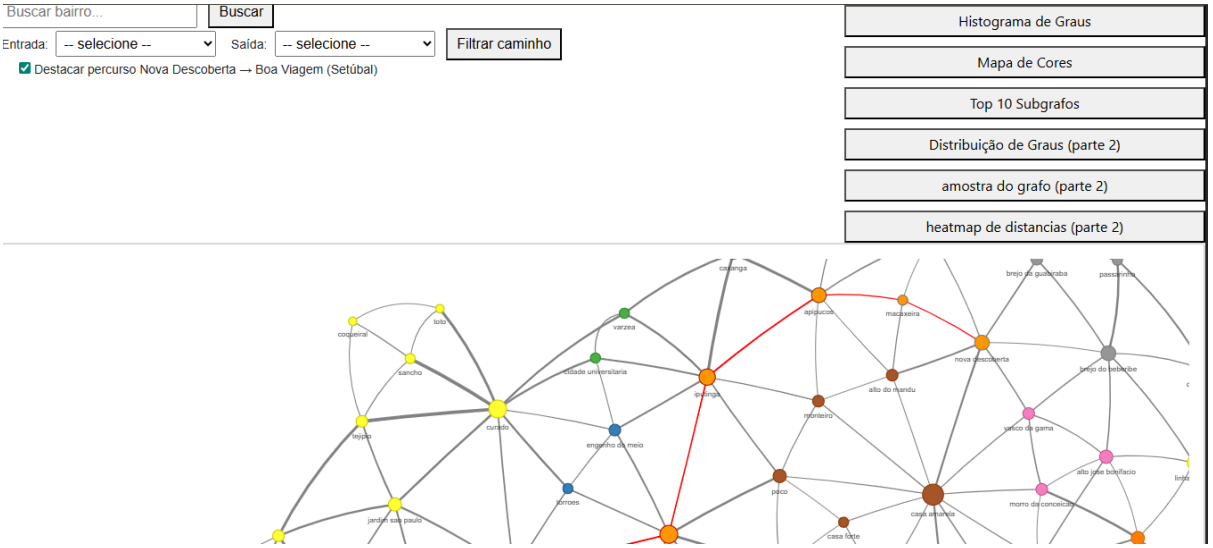
- Microrregiões muito densas indicam grupos de bairros intimamente conectados entre si.
- Microrregiões menos densas tendem a ser geograficamente mais fragmentadas.

Esse resultado permitiu que nós identificássemos regiões mais coesas e bairros que funcionam como “organizadores locais”.

3.5. Grafo Interativo

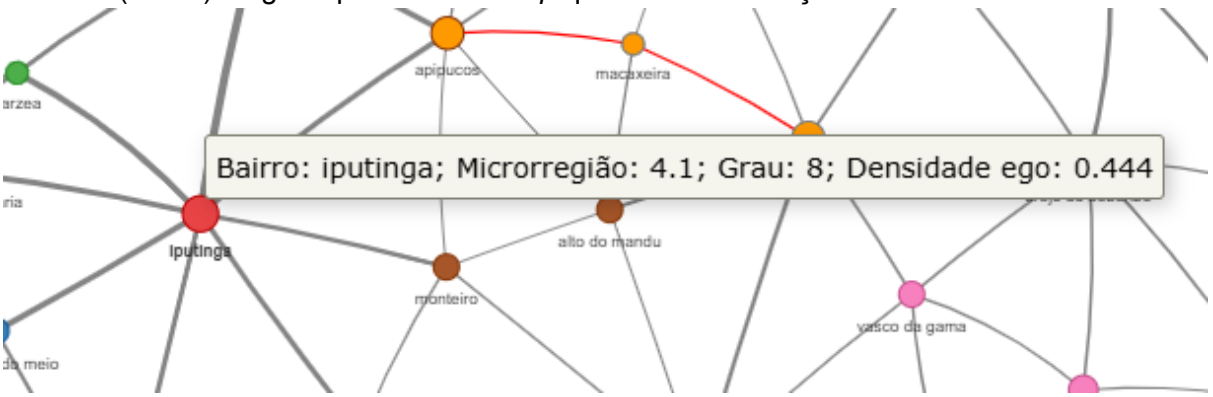
A função **gerar_grafo_interativo()** implementa um grafo interativo com recursos de visualização e interação. Ela utiliza a biblioteca **PyVis** para criar a rede visual dos bairros e suas conexões, permitindo que o usuário interaja com o grafo diretamente no navegador.

Recursos do HTML Interativo:



Tooltip com Grau e Microrregião

Cada nó (bairro) no grafo possui um *tooltip* que exibe informações sobre o bairro incluindo:



- **Nome do bairro**.
- **Microrregião** à qual o bairro pertence.
- **Grau** do nó (número de conexões do bairro).
- **Densidade ego** do bairro (uma métrica que avalia a conectividade do bairro e seus vizinhos).
- Esses detalhes são apresentados quando o usuário passa o mouse sobre um nó, oferecendo uma visão rápida das características de cada bairro.

```python
    # tooltip e cor por microrregião
    for u, info in stats.items():
        tooltip = (
            f"Bairro: {info['label']}; "
            f"Microrregião: {info['microrregiao']}; "
            f"Grau: {info['grau']}; "
            f"Densidade ego: {info['densidade_ego']:.3f}"
        )
        net.add_node(
            u,
            label=info["label"],
            title=tooltip,
            value=info["grau"],
            color=original_node_colors.get(u, default_node_color),
        )

```

Busca por Bairro:

- O HTML gerado inclui uma caixa de busca que permite ao usuário procurar por um bairro específico no grafo. Quando um bairro é encontrado, o grafo centraliza e destaca esse bairro, facilitando a visualização.

- O código JavaScript implementa a função de busca:

```javascript
function searchNode() {
  var q = document.getElementById('searchBox').value.toLowerCase().trim();
  if (!q) return;

  var allNodes = nodes.get();
  var target = null;

  for (var i = 0; i < allNodes.length; i++) {
    var label = String(allNodes[i].label || allNodes[i].id).toLowerCase();
    if (label.indexOf(q) !== -1) {
      target = allNodes[i];
      break;
    }
  }

  if (target) {
    network.focus(target.id, {
      scale: 1.5,
      animation: {
        duration: 800,
        easing: 'easeInOutQuad'
      }
    });
    network.selectNodes([target.id]);
  } else {
    alert("Nenhum bairro encontrado para: " + q);
  }
}

```

Botão para Destacar o Caminho Obrigatório:

- A função também inclui um botão que permite ao usuário destacar o caminho entre os bairros "Nova Descoberta" e "Boa Viagem (Setúbal)", com a opção de ativar ou desativar o destaque desse percurso no grafo.

- Quando ativado, o caminho entre os dois bairros é mostrado de forma mais visível, alterando a cor dos nós e arestas no caminho e facilitando a visualização do percurso.

```
446     function filtrarEntradaSaida() {
447       var origem = document.getElementById('origemSelect').value;
448       var destino = document.getElementById('destinoSelect').value;
449       if (!origem || !destino) {
450         alert("Selecione um bairro de entrada e um de saída.");
451         return;
452       }
453       var path = shortestPath(origem, destino);
454       if (!path || path.length === 0) {
455         alert("Não foi encontrado caminho entre os bairros selecionados.");
456         return;
457       }
458       highlightPath(path);
459       network.fit({
460         nodes: path,
461         animation: {
462           duration: 800,
463           easing: 'easeInOutQuad'
464         }
465       });
466     }
467   </script>
468   """
```

3.6. Passo a passo de como rodar

Instalar os requirements utilizando: **pip install -r requirements.txt**

Primeira parte:

- Processar e normalizar bairros: **python -m src.graphs.io**
- Calcular métricas globais, microrregiões e ego: **python -m src.global_metrics_calculator**
- Gerar análises específicas:
    - **python -m src.get_bairro-grau**
    - **python -m src.get_bairro_maior_grau**
    - **python -m src.get_bairro_mais_denso**
- Executar Dijkstra para todos os endereços: **python -m src.cli --calc-enderecos**
- Gerar a árvore do percurso obrigatório: **python -m src.gerar_arvore_percurso**
- Gerar grafo interativo em HTML: **python -m src.cli --grafo-interativo**
- Visualizações : **python -m src.run_viz**
- Para visualizar o grafo interativo é necessário colocar o path nd arquivo **grafo_interativo.html** no navegador

4. Parte 2 - Ligações Rodoviárias e Hidroviárias 2016

4.1. Fonte de Dados

O dataset utilizado nesta etapa do projeto foi o “Ligações Rodoviárias e Hidroviárias 2016”, produzido pelo Instituto Brasileiro de Geografia e Estatística (IBGE). Ele descreve uma rede de transporte intermunicipal formada por conexões entre as sedes dos municípios brasileiros que possuem serviços regulares de transporte coletivo, tanto rodoviário quanto hidroviário. Cada registro representa uma ligação direta entre dois municípios, contendo informações como o tempo de deslocamento, o custo da passagem e a frequência das viagens.

Com base nessas informações, foi construído um grafo direcionado e ponderado, em que cada vértice corresponde a um município e cada aresta representa uma ligação de transporte entre duas cidades. O peso das arestas foi definido pelo tempo de deslocamento (em segundos) entre os municípios, permitindo análises de caminhos mínimos e eficiência de rotas. Assim, a rede modela o tempo necessário para se deslocar entre diferentes pontos do país, possibilitando avaliar como os algoritmos de busca e de menor caminho — como BFS, DFS, Dijkstra e Bellman–Ford — se comportam diante de um cenário real e de grande escala.

O grafo resultante possui **5.161 vértices**, representando os municípios brasileiros, e **65.639 arestas direcionadas**, correspondentes às conexões de transporte identificadas pelo IBGE. Trata-se de uma rede extensa e heterogênea, na qual a maioria dos municípios possui poucas ligações diretas, enquanto grandes centros urbanos, como São Paulo, Belo Horizonte, Belém e Manaus, concentram um número elevado de conexões, atuando como nós centrais (hubs) da rede.

4.2. Implementação dos Algoritmos

Na Parte 2, o algoritmo de **Dijkstra** continua sendo o mesmo implementado manualmente em **algorithms.py**, reaproveitando integralmente a lógica já usada na Parte 1, com fila de prioridade (heapq) e listas de adjacência definidas na classe Graph. A principal evolução está na estrutura de grafo: a classe passou a receber um parâmetro booleano direcionado, permitindo escolher entre grafos direcionados ou não direcionados com a mesma implementação. Dessa forma, o código da Parte 1 pôde ser reutilizado na Parte 2 apenas ajustando esse parâmetro e o carregamento dos dados, sem necessidade de reescrever o algoritmo de caminhos mínimos.

O algoritmo de **Bellman-Ford** também foi implementado manualmente em **algorithms.py**, reutilizando a mesma estrutura de grafo baseada em lista de adjacência definida em graph.py. A função principal **bellman_ford()** segue uma variação otimizada do Bellman-Ford no estilo SPFA (Shortest Path Faster Algorithm): em vez de percorrer todas as arestas em todas as iterações, o algoritmo mantém uma fila de nós a serem processados e só relaxa arestas a partir dos vértices cuja distância acabou de melhorar. As distâncias são

armazenadas em dist, os predecessores em pred, e um contador por nó é usado para detectar ciclos negativos; quando um nó é atualizado vezes demais, é lançada a exceção NegativeCycle(). As funções públicas **bellman_ford_path()** e **bellman_ford_path_length()** usam esse núcleo para retornar, respectivamente, o percurso e o custo mínimo entre origem e destino, tratando também os casos de vértices inexistentes (NodeNotFound) ou ausência de caminho (NoPath).

Na função **calcular_distancias_bellman_ford()** o grafo de municípios é carregado como grafo direcionado a partir do dataset da Parte 2, e o algoritmo é aplicado para todos os pares origem–destino definidos em **enderecos_bellmanford.csv**, gerando arquivos CSV e JSON com os resultados. Para testar especificamente o comportamento frente a pesos e ciclos negativos, foram modificadas três linhas do dataset, introduzindo arestas artificiais com tempo negativo que formam, por exemplo, o ciclo Palmitos → São Carlos → São José do Rio Preto → Palmitos. Esses testes permitem validar a detecção de ciclos negativos pela implementação, embora, no contexto real do problema, pesos negativos não façam sentido, já que o peso representa tempo de deslocamento entre municípios.

Além dos algoritmos de caminhos mínimos, foram implementadas também as **buscas em largura (BFS – Breadth-First Search)** e **em profundidade (DFS – Depth-First Search**), adaptadas para o grafo direcionado construído a partir do dataset de ligações rodoviárias e hidroviárias. Ambas as funções foram implementadas manualmente no arquivo **algorithms.py**, reutilizando a mesma estrutura de grafo baseada em lista de adjacência da classe Graph, definida em **graph.py**.

No caso da BFS, a função **bfs_ordem_camadas_ciclos_dir()** recebe como entrada o grafo direcionado e um vértice de origem (um município) e realiza uma exploração em largura a partir desse nó. Internamente, o algoritmo utiliza uma fila para controlar os vértices a serem visitados, um conjunto visited para registrar quais nós já foram explorados, um dicionário parent para armazenar o predecessor de cada vértice na árvore de busca e um dicionário dist para guardar a “distância em número de arestas” entre a origem e cada nó alcançado. Como o grafo é direcionado, a BFS considera apenas as arestas no sentido v → u retornadas por G.vizinhos(v).

A função também identifica possíveis ciclos direcionados a partir de arestas não pertencentes à árvore de busca. Para isso, quando encontra uma aresta para um nó já visitado que não é o pai direto do vértice atual, o algoritmo reconstrói os caminhos da origem até os dois nós envolvidos, encontra o ancestral comum mais próximo e monta uma lista de vértices que compõe o ciclo. Esses ciclos são armazenados em uma lista de cycles até um limite configurável (max_cycles) e deduplicados com o auxílio do conjunto seen_cycles. Ao final, a BFS retorna três estruturas principais: ordem (ordem de visita dos nós), camadas (lista de listas com os vértices por nível de distância a partir da origem) e ciclos (lista de ciclos direcionados detectados).

A busca em profundidade foi implementada na **função dfs_ordem_camadas_ciclos_dir()**, também adaptada para o grafo direcionado. Em vez de utilizar recursão, que poderia estourar o limite de profundidade do Python devido ao tamanho da rede, implementamos utilizando uma pilha contendo pares (nó_atual, iterador_de_vizinhos), simulando o comportamento recursivo de forma iterativa. Assim como na BFS, são mantidas as estruturas visited (nós já visitados), parent (predecessores),

com a adição de depth (profundidade ou nível da árvore de DFS a partir da origem).

Durante a execução, a DFS empilha um vizinho ainda não visitado sempre que encontra uma aresta de árvore e continua aprofundando até esgotar os vizinhos daquele ramo, voltando pela pilha quando atinge um vértice sem novos destinos. Para detecção de ciclos, a função segue a mesma ideia da BFS: quando encontra uma aresta para um nó já visitado que não é o pai direto, reconstrói os caminhos até a raiz, encontra o ancestral comum mais próximo e monta o ciclo, respeitando o limite de max_cycles e evitando duplicidades. Ao final, a função retorna a ordem de visita 'ordem', as camadas construídas a partir da profundidade e a lista de ciclos 'ciclos'.

Na Parte 2, as funções bfs() e dfs() definidas em **solve.py** são responsáveis por integrar essas implementações ao fluxo do projeto. Ambas carregam o grafo direcionado a partir do arquivo **LRH2016_00_Base_Completa.csv**, construindo os vértices e arestas de acordo com as ligações intermunicipais do IBGE, e em seguida leem o arquivo **enderecos_parte2.csv**, que especifica pares de municípios de interesse (campo municipio_inicio e municipio_destino). Para cada linha desse arquivo, o script executa a BFS ou a DFS a partir do município de origem e salva um arquivo JSON na pasta out/parte2/BFS ou out/parte2/DFS, respectivamente, contendo a fonte, o destino, a ordem de visita, as camadas e os ciclos encontrados.

4.3. Resultados e Desempenho

4.3.1 Testes e Métricas

Para a parte 2 realizamos nossos testes sobre um grafo dirigido com **4533 nós** e **65.556 arestas**, utilizando quatro algoritmos distintos: **BFS, DFS, Dijkstra e Bellman-Ford**. Além disso, selecionamos as métricas de: tempo necessário para executar o caminho, memória gasta, tempo por distância e memória consumida por distância, utilizando, para isso, as distâncias de cada percurso.
Para rodá-las, é necessário só utilizar o comando ***python -B -m tests.metrics_pt2***
Para cada algoritmo, testamos com as mesmas 3 origens, conforme pode-se conferir abaixo:

4.3.1.1 BFS

- Para o BFS, assim como para os demais, avaliamos três origens: Santo Antônio do Sudoeste, Planura e Wanderlândia.

  A execução a partir de Santo Antônio do Sudoeste levou aproximadamente **0,0778 segundos**, utilizando cerca de **0,49 MB de memória de pico**, visitando **4251 nós** ao longo de **9 camadas**, com detecção de **10 ciclos**.
  À partir de Planura, o algoritmo executou em **0,0701 segundos**, com memória de pico de **0,41 MB**, também visitando **4251 nós**, distribuídos em **10 camadas**, e identificando **10 ciclos**.
  Já iniciando por Wanderlândia, o BFS foi ainda mais rápido, com **0,0562 segundos** de execução e **0,41 MB de memória**, novamente visitando **4251 nós**, com **9 camadas** e **10 ciclos**.

- De modo geral, o BFS percorreu praticamente todo o grafo nas três execuções, com tempo muito baixo e uso de memória bastante estável.

4.3.1.2 DFS

- O DFS apresentou tempos extremamente reduzidos, porém visitou apenas frações pequenas do grafo, por seguir um aprofundamento único antes de ramificar.
  A partir de Santo Antônio do Sudoeste, o tempo foi de apenas **0,00192 segundos**, com memória de **0,04 MB**, visitando somente **27 nós**, distribuídos em **15 camadas**, e detectando **10 ciclos**.
  A partir de Planura, o tempo foi de **0,00175 segundos**, com o mesmo uso de memória de **0,04 MB**, visitando **35 nós** em **22 camadas**, com **10 ciclos**.
  Já partindo de Wanderlândia, a execução levou **0,00119 segundos**, com **0,04 MB**, visitando **34 nós**, também distribuídos por **22 camadas**, e detectando **10 ciclos**.
- Ou seja, o DFS foi o algoritmo mais rápido, mas explorou caminhos muito limitados, tornando-se adequado para análises estruturais, e não para cobrir todo o grafo.

4.3.1.3 Dijkstra

- O Dijkstra apresentou desempenho bastante variado conforme o par de origem e destino.
  Da origem Santo Antônio do Sudoeste para Planura, o cálculo do caminho levou **0,253 segundos**, com **0,80 MB de memória**, enquanto o cálculo da distância levou **0,190 segundos** com **0,73 MB**, resultando em uma distância total de **1086** e um caminho com **12 etapas**.
  Da mesma origem para Wanderlândia, o cálculo do caminho executou em **0,116 segundos** com **0,62 MB**, e o cálculo da distância em **0,124 segundos** com memória igual. A distância encontrada foi **721**, com **8 etapas no caminho**.
- Partindo de Planura para Santo Antônio do Sudoeste, o tempo foi muito reduzido: **0,0064 segundos** para o caminho e **0,0075 segundos** para a distância, ambos utilizando apenas **0,09 MB**, com distância de **–850** e caminho de **8 etapas**.
  De Planura para Wanderlândia, as execuções levaram **0,0585 segundos** e **0,0548 segundos**, com uso de **0,46 MB**, gerando distância de **–440** e caminho de **10 etapas**.
- A partir de Wanderlândia para Santo Antônio do Sudoeste, o cálculo do caminho levou **0,0357 segundos**, usando **0,33 MB**, enquanto a distância levou **0,0297 segundos**, com distância de **800** e caminho com **11 etapas**.
  Para o trajeto Wanderlândia → Planura, os tempos foram **0,1089** e **0,1343 segundos**, ambos com **0,83 MB**, e a distância encontrada foi **710**, com **25 etapas no caminho**.
- O Dijkstra variou entre execuções extremamente rápidas (milissegundos) e outras mais pesadas (até 0,25 segundos), com consumo de memória de 0,09 a 0,86 MB. Essas variações se dão pelo fato de que o Dijkstra não foi feito para processar pesos negativos.

4.3.1.4 Bellman-Ford

- O Bellman-Ford apresentou tempos maiores, porém uso de memória mais constante.
  No trajeto Santo Antônio do Sudoeste → Planura, o cálculo do caminho levou **0,262 segundos**, com **0,55 MB**, enquanto o cálculo da distância levou **0,280 segundos**, resultando em distância de **1665** em **7 etapas**.
  Para Santo Antônio do Sudoeste → Wanderlândia, o algoritmo identificou **um ciclo negativo**, impossibilitando o cálculo da menor distância.
- Partindo de Planura para Santo Antônio do Sudoeste, o algoritmo levou **0,162 segundos** para o caminho e **0,181 segundos** para a distância, com distância de **–845** em **6 etapas**.
  Da mesma origem para Wanderlândia, os tempos foram **0,364 segundos** e **0,437 segundos**, com distância de **–1350** em **6 etapas**.
- A partir de Wanderlândia para Santo Antônio do Sudoeste, o Bellman-Ford executou em **0,243 segundos** no cálculo do caminho e **0,188 segundos** no cálculo da distância, resultando em distância de **295** com **6 etapas**.
  Para Wanderlândia → Planura, os tempos foram **0,261 segundos** e **0,298 segundos**, com distância de **1528** e novamente **6 etapas**.
- Um ponto crítico foi a detecção do ciclo negativo formado por São Carlos, São José do Rio Preto, Palmitos, Lajeado, Nova Olinda e Wanderlândia, com custo total de **–515**, algo que apenas o Bellman-Ford é capaz de identificar entre os algoritmos avaliados.

Análise:

- Mesmo mais lento (0.16–0.44 s), o algoritmo foi mais estável em consumo de memória (sempre ~0.55 MB).

- Identificou corretamente um **ciclo negativo**, recurso que Dijkstra não possui, sendo mais adequado entre os demais.

4.4. Discussão Crítica

A partir dos resultados observados, foi possível analisar o desempenho de cada um e determinar qual é de fato mais adequado para o nosso cenário.

4.4.1 Eficiência e aplicabilidade

4.4.1.1 BFS

Mais eficiente para travessias amplas. Seria a melhor escolha para explorarmos o grafo percorrendo ele todo, verificar conectividade ou calcular distâncias em grafos sem pesos.

4.4.1.2 DFS

Extremamente rápido, mas visita poucas rotas, sendo mais adequado para detecção de ciclos e estruturação do grafo, não para medir distâncias.

4.4.1.3 Dijkstra

Rápido e eficiente para pesos não negativos, porém não para o nosso contexto, uma vez que não foi feito para processar pesos negativos, tendo desempenho instável e consumindo muita memória.

4.4.1.4 Bellman-Ford

Apesar de mais lento, foi o único que detectou corretamente o ciclo negativo presente, sendo **o mais adequado para nosso contexto.** Além disso, consumiu menos memória e foi mais estável quando comparado com Dijkstra.

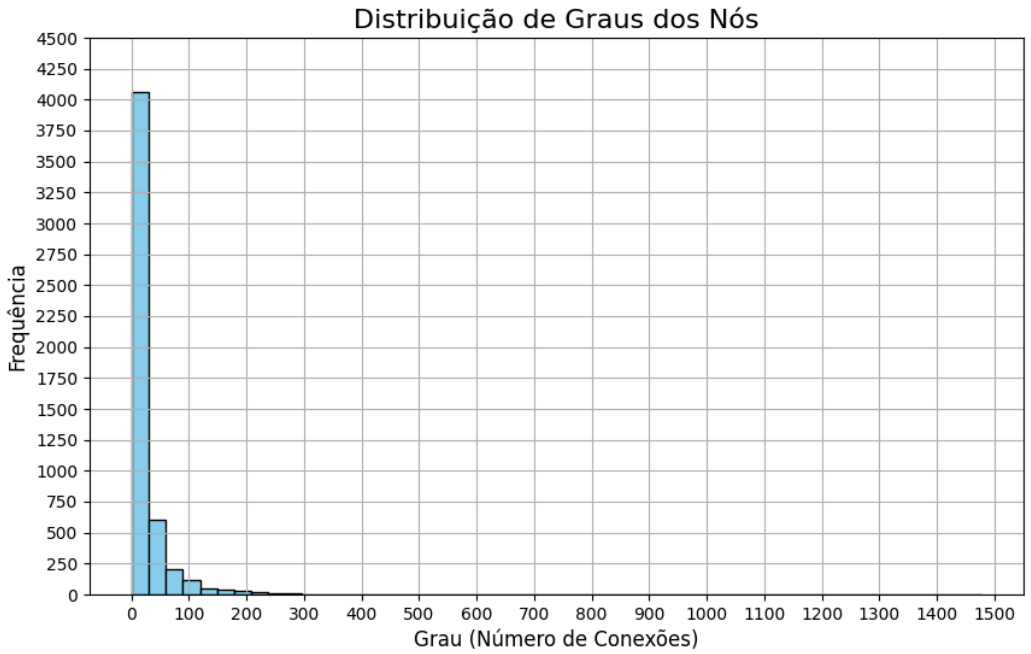
4.5. Visualizações da Parte 2

1. Distribuição de Graus

A **Distribuição de Graus** é uma visualização no formato de um **histograma**, que mostra a frequência com que os municípios (ou nós) têm determinado número de conexões (arestas). O grau de um nó é simplesmente o número de arestas que o conectam a outros nós.

- **O que essa visualização revela**:
    - **A densidade de conexões**: Ao observar o histograma, podemos perceber se a rede é homogênea ou se existem nós com um número muito grande de conexões em comparação aos outros. Por exemplo, se poucos nós têm muitos vizinhos e a maioria tem poucos, isso indica uma rede com alto grau de centralização.
    - **A variedade de conectividade**: O histograma também pode mostrar se a distribuição de graus segue uma tendência de "pico" em torno de certos valores, ou se ela é mais dispersa, indicando uma rede mais equilibrada em termos de conectividade.

Essa visualização é importante para avaliar a **centralidade** na rede, identificar nós "hubs" (com muitas conexões) e observar a **uniformidade** das conexões entre nós.

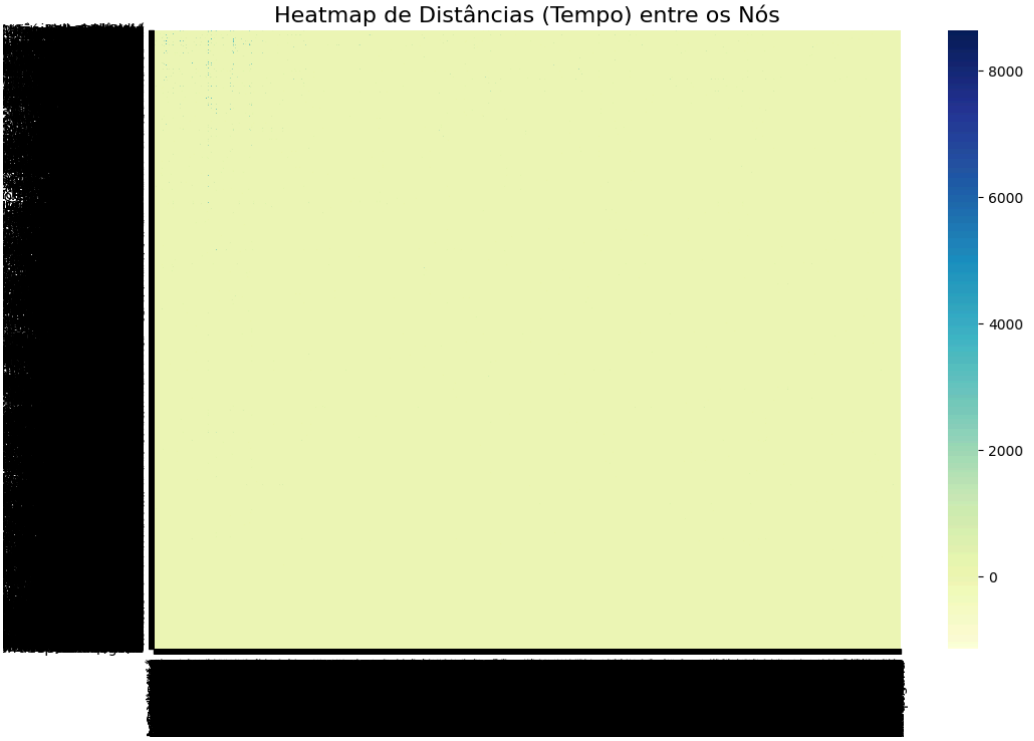


2. Heatmap de Distâncias

O **Heatmap de Distâncias** é uma representação visual das distâncias entre todos os pares de nós da rede. Cada célula do mapa de calor corresponde ao valor da distância (ou tempo de viagem, dependendo do contexto) entre dois municípios. Quanto mais escura a célula, maior é a distância entre nós.

- **O que essa visualização revela**:
    - **Padrões de proximidade e distância**: O heatmap facilita a visualização das relações de proximidade entre diferentes bairros ou municípios. Ele permite identificar rapidamente quais municípios estão mais próximos entre si e quais estão mais distantes.
    - **Identificação de clusters**: Pode ser fácil ver agrupamentos de municípios que têm distâncias mais próximas, formando regiões ou zonas com características semelhantes.
    - **Análise de custos de deslocamento**: O heatmap ajuda a identificar quais trajetos ou conexões exigem mais tempo ou custo para serem percorridos, o que pode ser útil em análises de eficiência de transporte, planejamento urbano ou otimização de rotas.

Essa visualização é extremamente útil para **analisar as relações de distância** de forma clara e rápida, apesar de a grande extensão da planilha fonte ter sido prejudicial para a visualização, mesmo assim ajudando a identificar padrões de proximidade ou áreas que exigem maior atenção no planejamento de rotas ou melhorias na infraestrutura.



3. Amostra do Grafo Maior

A **Amostra do Grafo Maior** é uma visualização interativa que representa as conexões entre os nós (municípios ou bairros) de um grande conjunto de dados. No grafo, cada nó é um município e cada aresta é uma conexão entre dois municípios, com o peso representando, por exemplo, o tempo de viagem ou a distância entre eles.

- **O que essa visualização revela**:
    - **Estrutura global do grafo**: Permite observar como os municípios estão conectados entre si, formando uma rede de transporte ou interação.
    - **Identificação de nós centrais**: Alguns municípios podem se destacar por ter mais conexões (arestas), o que indica sua importância ou centralidade na rede.
    - **Interatividade**: O usuário pode explorar o grafo, ampliando, diminuindo ou focando em áreas específicas, facilitando a análise visual das relações entre os municípios. O uso de cores, tamanhos de nós e espessura de arestas

pode ajudar a identificar rapidamente padrões de conectividade.

Essa visualização é útil para quem deseja entender as principais conexões de uma rede e explorar as interações entre os elementos de forma intuitiva.

4.6. Passo a passo de como rodar

Segunda parte:

- Rodar BFS direcionado: **python -m src.cli --bfs**
- Rodar DFS direcionado: **python -m src.cli --dfs**
- Executar testes de Dijkstra + Bellman–Ford: **python -B -m tests.cli_parte2 --calc-enderecos-parte2**