CESAR SCHOOL

TEORIA DOS GRAFOS

RELATÓRIO

Antônio Paulo Araújo de Barros e Silva

Clara Machado de Araújo

Lisa Matubara

Luziane Pires dos Santos

1. Introdução

O presente trabalho tem por objetivo aplicar os conhecimentos adquiridos na disciplina Teoria dos Grafos e está dividido em duas partes. A primeira parte consiste na representação dos bairros da cidade do Recife como um grafo, onde cada bairro configura um vértice e as ruas, avenidas ou pontes que ligam esses bairros são as arestas. Partindo dessa estrutura, o objetivo é analisar como os bairros se conectam, gerando análises a partir das métricas estabelecidas com base nessas conexões.

Por sua vez, a segunda parte do trabalho consiste na escolha de um *dataset* distinto para aplicar os algoritmos clássicos de grafos, com o objetivo de explorar caminhos e comparações de desempenho. Para tanto, serão implementados e analisados os algoritmos de Dijkstra, Bellman-Ford, BFS e DFS, verificando tanto a exatidão dos resultados quanto o tempo de execução de cada método, permitindo observar o comportamento dos algoritmos mencionados.

2. Grafo dos bairros de Recife

a. Derretimento, limpeza e padronização dos dados

A primeira etapa do trabalho envolveu o "derretimento" do arquivo *.csv* disponibilizado nas orientações, o qual continha os bairros da cidade do Recife e suas respectivas microrregiões. O arquivo *bairros_recife.csv*, fornecido originalmente em formato de planilha larga (com colunas representando microrregiões), foi "derretido" com a função *pd.melt()* para gerar uma lista única de bairros[1].

O processo de limpeza e padronização dos dados, realizado de forma automática (via código) e por alterações manuais, eliminou valores ausentes

[1] A primeira etapa foi construída no arquivo *io.py*, cuja função *_processar_e_salvar_bairros() lê* o arquivo original (*bairros_recife.csv*), faz o derretimento através da função *pd.melt() d*a biblioteca *pandas*, e transforma as colunas de microrregiões em linhas, gerando uma lista onde cada bairro está associado à sua respectiva microrregião. Outras funções do *pandas* foram utilizadas para a limpeza dos dados, como a remoção de valores vazios, duplicados e espaços em excesso, resultando na organização dos bairros em ordem alfabética, todas presentes no mencionado arquivo *io.py*.

e duplicados, padronizou os nomes e utilizou o padrão minúsculo e sem acentos para facilitar as etapas subsequentes, resultando no arquivo *bairros_unique.csv*[2], cuja divisão em microrregiões foi verificada junto ao site da Prefeitura do Recife[3].

Em seguida, tomando por base a lista de bairros constantes no arquivo gerado na etapa anterior, foi criado manualmente o arquivo *adjacências_bairros.csv*, contendo, em cada linha, um par de bairros que possuem ligação direta, bem como o logradouro[4] que faz a conexão entre eles e o peso atrelado a essa conexão, o qual foi estipulado pela equipe de acordo com o tipo da via: travessa/ladeira (1.0), viaduto/ponte (2.0), rua/estrada (3.0), avenida (4.0) e BR (5.0).

b. Análise das métricas

Com os dados ajustados, iniciou-se a elaboração do código necessário para as análises. Essa etapa foi implementada no arquivo *solve.py*, o qual inicialmente faz a chamada da função *construir_grafo_principal()*, que é responsável por montar o grafo principal do projeto através das seguintes etapas:

- carregamento dos dados, feito por meio da chamada da função *carregar_dados_principais()*, constante do arquivo *io.py* e que carrega os *dataframes df_bairros* (baseado em *bairros_unique.csv*) e *df_adjacencias* (baseado em *adjacências_bairros.csv*);
- criação da estrutura do grafo, através da instanciação de um objeto vazio, da classe Grafo (definida em *graph.py*), que servirá para armazenar os nós (bairros) e as arestas (conexões entre bairros) e

[2] Para facilitar as visualizações e análises, e seguindo orientações obtidas durante o acompanhamento do projeto, o bairro de Setúbal foi incluído nessa lista, estando conectado à Boa Viagem.
[3] A esse respeito, consultar https://www2.recife.pe.gov.br/servico/perfil-dos-bairros.
[4] Para os fins deste trabalho, entende-se como logradouro a conexão existente entre o par de bairros considerado, podendo ser uma rua, avenida, ponte, etc., ou seja, qualquer forma de ligação viária entre os bairros que permita a passagem de veículos, obtida a partir do mencionado site sobre o Perfil dos Bairros, bem como por consultas ao Google Maps.

pesos atrelados a elas, usando, respectivamente as funções *add.node()* e *add.edge()*;

- retorno do grafo completo, bem como dos *dataframes* utilizados, além da exibição da quantidade de nós e arestas criados.

Em seguida, a análise do grafo principal é feita pela função *analisar_grafo_completo()*, valendo-se da função auxiliar *_calcular_metricas_basicas()*, a qual é responsável por calcular três medidas fundamentais: ordem (quantidade de bairros), tamanho (número de conexões) e densidade (grau de conexão entre os bairros), obtidas da seguinte forma[5]:

- a ordem é calculada por meio da função *grafo.get_numero_de_nos()*[6];
- o tamanho utiliza a função *grafo.get_numero_de_arestas()*[7];
- a densidade vale-se tanto do número de bairros (nós) quanto do número de arestas (conexões), sendo calculada por meio da seguinte fórmula:

$$D = \frac{2M}{N(N-1)}$$

$$M = número\ de\ conexões\ e\ N = número\ de\ bairros$$

As análises das métricas referentes às microrregiões é feito pela função *analisar_microrregioes()*, que igualmente se vale da função auxiliar *_calcular_metricas_basicas()* para calcular a ordem, o tamanho e a densidade dos subgrafos formados por cada microrregião.

A execução do código mostrou que o grafo possui 95 bairros e 244 ligações, sendo o resultado (dicionário de métricas) armazenado no arquivo *recife_global.json*. Da mesma forma, o processo é repetido, mas de forma

---

[5] Ressalte-se que a função *_calcular_metricas_basicas()* retorna um dicionário: {‘ordem’, ‘tamanho’, ’densidade’}.

[6] Dentro da classe Grafo, essa função retorna *len(self.adj)*, ou seja, a quantidade de chaves no dicionário de adjacência, que corresponde ao número de bairros (vértices) do grafo. Em outras palavras, a ordem é calculada contando quantos bairros diferentes existem no grafo.

[7] Ainda na classe Grafo, existe um atributo *self.num_arestas* que começa em 0 e é incrementado em 1 toda vez que *add_edge* é chamada para inserir uma conexão entre dois bairros. Depois, *get_numero_de_arestas()* apenas retorna esse contador, ou seja, o tamanho é calculado contando quantas conexões (arestas) foram adicionadas ao grafo a partir do arquivo de adjacências.

individualizada para cada subgrafo que representa uma microrregião constante no *dataframe df_bairros* salvando-se o dicionário de métricas no arquivo *microrregioes.json*.

c. Ego-rede por bairro

No que se refere à análise da ego-rede, a função *analisar_ego_redes()* identifica os vizinhos diretos de cada bairro (com o método *get_vizinhos(v)*) e define o grau e a quantidade desses vizinhos, construindo, em seguida, a ego-rede. Para cada ego-rede, formada pelo próprio bairro e seus vizinhos, são realizados os cálculos de ordem, tamanho e densidade[8], armazenando os dados no arquivo *ego_bairro.csv*.

d. Graus e rankings

Na análise de graus e ranking, a função *analisar_graus_e_ranking()* recebe o grafo principal e, para cada bairro, calcula a quantidade de conexões diretas que ele possui com outros bairros, ou seja, quantas ligações (arestas) partem dele no grafo, o que representa o grau (número de vizinhos) de cada bairro (vértice), valendo-se, para tanto, das funções *get_vizinhos(bairro)* e *len(vizinhos)*[9].

A utilização da função *idxmax()* no *dataframe df_graus*, (criado na listagem dos graus) identificou o bairro de Casa Amarela como o de maior grau, com 11 conexões, o que reflete sua característica de ser um dos bairros mais conectados da cidade.

Por fim, para identificar o bairro mais denso, considera-se o número de bairros presentes na ego-rede (incluindo o bairro de referência) e o número

[8] A ordem é calculada a partir do número de bairros da ego-rede. Para o cálculo do tamanho, o arquivo de adjacências é percorrido para verificar a quantidade de conexões existentes entre os bairros que pertencem a esse mesmo grupo. Por sua vez, a densidade é calculada da mesma forma mencionada no tópico anterior, mas aplicada ao subgrafo da ego-rede.

[9] O resultado dessa etapa consiste na montagem do *dataframe df_graus* com a listagem dos graus, armazenando-se os resultados no arquivo *graus.csv*.

de ligações existentes entre eles, e, para isso, utiliza-se novamente a função *idxmax()* no campo *densidade_ego* (integrante do arquivo *ego_bairro.csv*).

De acordo com os resultados obtidos, os bairros Brasília Teimosa, Caçote, Hipódromo, Ipsep, Pau-Ferro, Setúbal, Sítio dos Pintos e Soledade apresentaram densidade máxima (1.0). Nos casos de Setúbal e Pau-Ferro, isso acontece porque eles possuem apenas um vizinho (Boa Viagem e Guabiraba, respectivamente). Nos demais casos, a densidade máxima indica que os bairros vizinhos também se conectam entre si, formando redes bem integradas dentro do grafo da cidade.

e. Distâncias entre endereços (Dijkstra)

Para o cálculo da distância entre os endereços foi utilizado como fonte o arquivo *endereços.csv*, que contém cinco pares de bairros aleatoriamente escolhidos pela equipe, incluindo-se, entre eles, o par obrigatório “nova descoberta, setubal”.

Após a normalização, certificando-se do uso de minúsculas e eliminando eventuais espaços em branco ainda existentes, os pares passam pelo algoritmo de Dijkstra para calcular a distância e o custo atrelado a ela, salvando-se os resultados no arquivo *distancias_enderecos.csv* (e, para o par obrigatório, também em *percurso_nova_descoberta_setubal.json*).

Para o cálculo do caminho e das distâncias dos pares, foram utilizadas, respectivamente, as funções *dijkstra_path()* e *dijkstra_path_lenght()*, responsáveis por aplicar o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois vértices em um grafo ponderado e cuja construção foi obtida a partir das referências constantes no site NetworkX[10].

[10] Para mais detalhes, consultar https://networkx.org/documentation/stable/_modules/networkx/algorithms/shortest_paths/weighted.html#dijkstra_path.

No projeto, estas funções estão definidas no arquivo *algorithms.py* e são usadas dentro da função *calcular_distancias_enderecos()*, em *solve.py*, para calcular as rotas entre os pares de bairros considerados. Nesse sentido, enquanto a função *dijkstra_path()* retorna o percurso completo, ou seja, a sequência de bairros que compõe o caminho mais curto entre o bairro de origem e o de destino, a função *dijkstra_path_leght()* percorre o grafo comparando os pesos das arestas, escolhendo sempre o menor custo acumulado, ou seja, a soma dos pesos das arestas percorridas no trajeto encontrado pela função anterior.

A título de exemplo, o percurso entre os bairros Nova Descoberta e Setúbal implica em um custo total de 27.0, obtido a partir do seguinte trajeto: *nova descoberta -> alto do mandu -> monteiro -> iputinga -> cordeiro -> prado -> afogados -> imbiribeira -> boa viagem -> setubal*, resultado que se coaduna com os dados constantes nos arquivos-base utilizados no projeto.

f. Transformar o percurso em árvore

Para a construção da visualização da árvore de percurso, foi desenvolvida a função *exportar_arvore_percurso_png()*[11] com o auxílio da biblioteca *matplotlib*. Essa etapa tem como objetivo representar o caminho encontrado entre os bairros de Nova Descoberta e Setúbal, calculado anteriormente pelo algoritmo de Dijkstra.

A função recebe o percurso e cria uma figura no formato horizontal, onde cada vértice corresponde a um bairro existente no trajeto e cada aresta representa a ligação entre eles. Cada bairro é representado por um círculo (*plt.scatter*), sendo o início (Nova Descoberta) colorido em azul, o fim (Setúbal) em vermelho e os intermediários em verde-claro.

[11] Esta função foi implementada no arquivo *viz.py* e é chamada a partir de *solve.py*, responsável por executar as etapas do projeto.

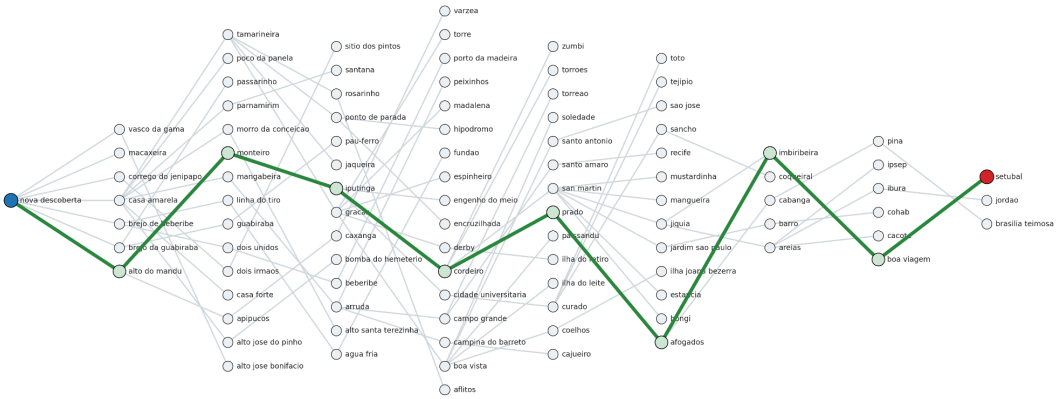
Acima de cada vértice, a função adiciona o nome do bairro com *plt.text*, garantindo a identificação de cada ponto do trajeto. Por fim, a figura, uma visualização estática do percurso mencionado, é formatada e o resultado é exportado para o arquivo *arvore_percurso.png*.

g. Explorações e visualizações analíticas

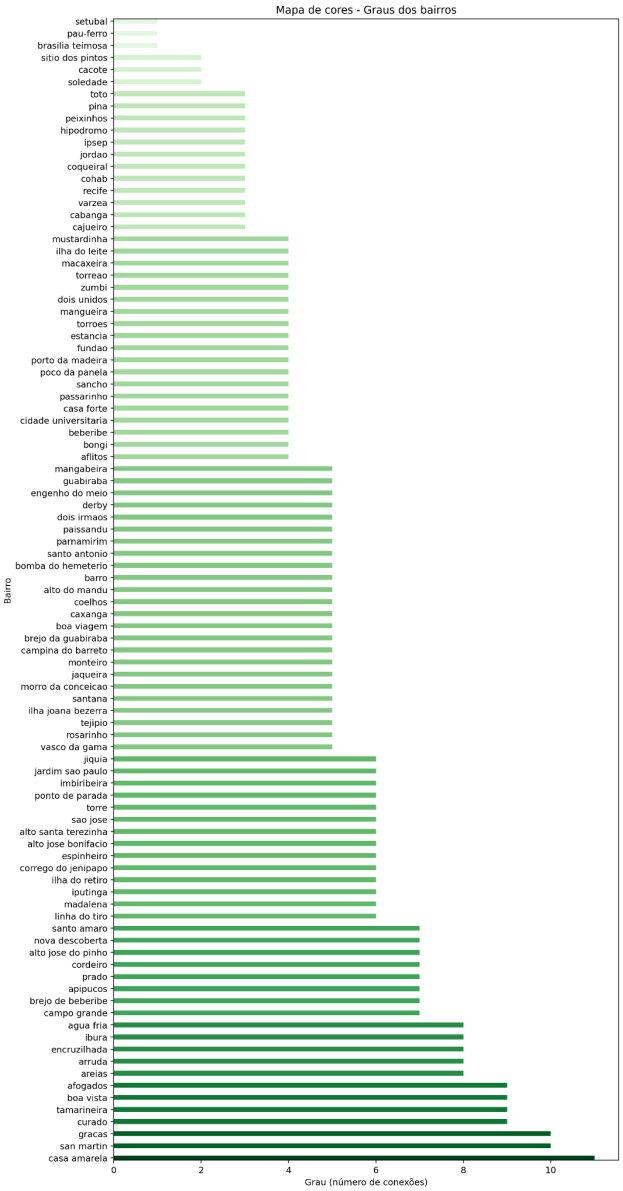
As implementações relacionadas às explorações e visualizações analíticas foram quase que integralmente reunidas na função *exploracoes_visuais()*, que, definida no arquivo *solve.py*, recebe o grafo principal e o *dataframe* de graus e chama as rotinas de visualização implementadas em *viz.py*.

A primeira delas é a função *exportar_arvore_percurso_destacada()*, que organiza os bairros em uma estrutura em árvore a partir da raiz “nova descoberta” e salva a figura gerada em *out/arvore_percurso_destacada.png*. Essa função aproveita o caminho obtido no item anterior e monta uma árvore em camadas tendo “nova descoberta” como raiz[12]. A imagem gerada mostra todos os bairros organizados por nível de distância, ligados por arestas em cinza claro, enquanto o percurso obrigatório aparece destacado em verde, com o início em azul e o fim em vermelho. Essa visualização permite enxergar onde o caminho se encaixa dentro da estrutura geral do grafo.

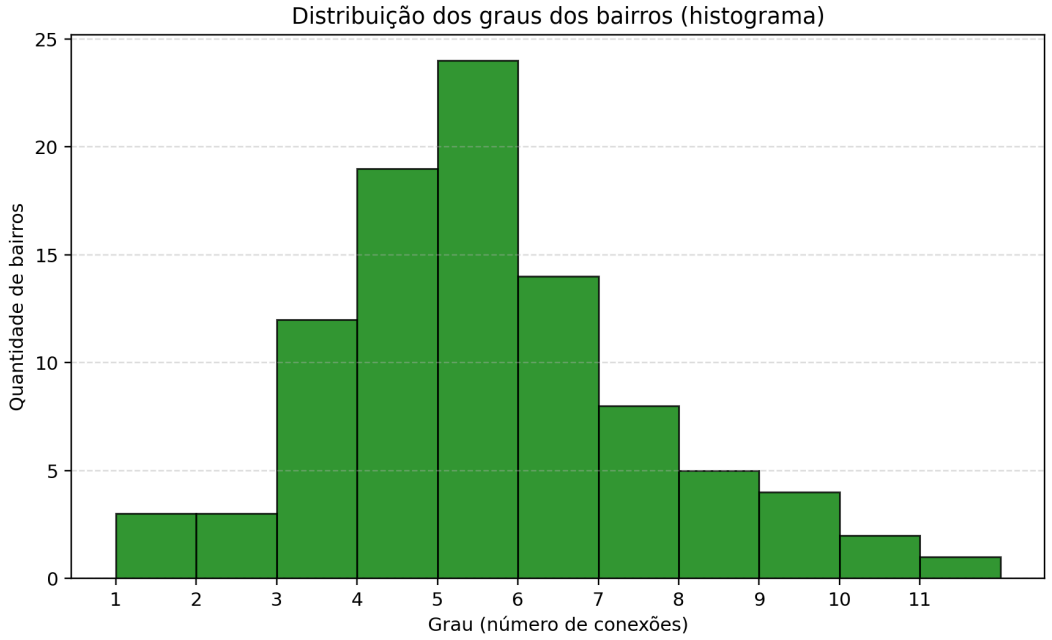
[12] Para que o caminho calculado no item anterior (alínea *f*) não se perca (já que ele não é armazenado fora dessa etapa), a geração dessa árvore é feita diretamente dentro da função *gerar_arvore_percurso(grafo)* em *solve.py*, logo após o cálculo do percurso.



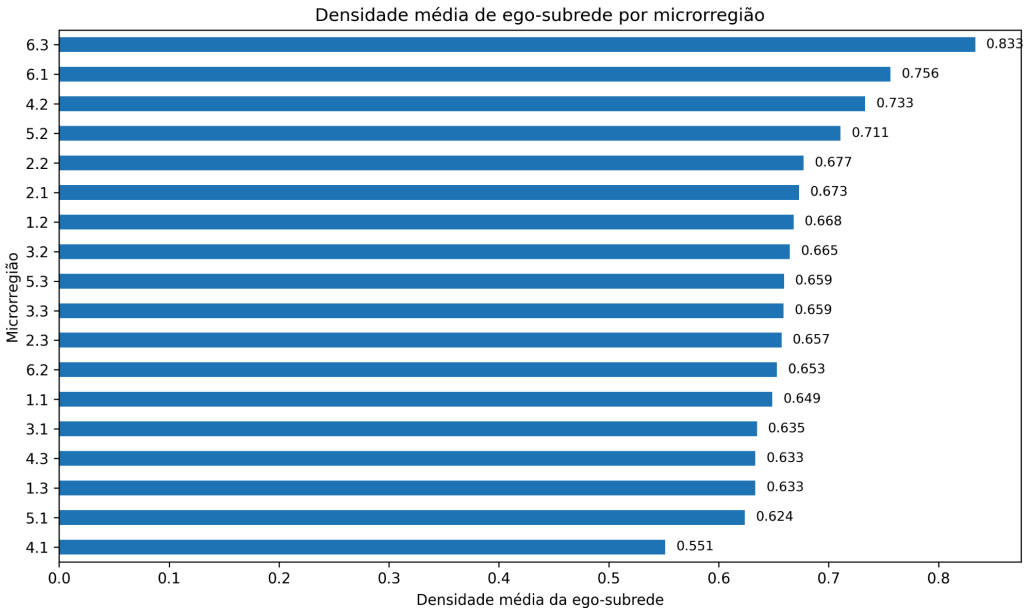
Por sua vez, a função *mapa_cores_por_grau()* utiliza a lista de graus gerada na alínea *d* e cria um gráfico de barras horizontais em que cada linha representa um bairro e o comprimento da barra indica o número de conexões (grau) daquele vértice, usando uma escala de cores em tons de verde para destacar os valores mais altos. Para tanto, a função pega o *dataframe df_graus* e organiza os bairros em ordem decrescente de grau. Em seguida, ela normaliza esses valores dividindo cada grau pelo maior grau encontrado, para obter números entre 0 e 1. Com o auxílio da biblioteca *matplotlib* (*plt.cm.Greens*), ela transforma esses valores normalizados em diferentes tons de verde e monta um gráfico de barras horizontais em que cada linha representa um bairro e a extensão da barra corresponde ao seu grau. Ao final, o gráfico é salvo como *out/mapa_cores_grau.png*.



A função *histograma_graus()* começa lendo a coluna grau do *dataframe df_graus*. A partir desses valores, ela identifica os graus mínimo e máximo presentes na rede e, a partir disso, cria uma sequência contínua de valores inteiros entre esses dois extremos. Esses valores formam os *bins* do histograma, garantindo que cada possível grau tenha sua própria barra no gráfico, sem pular faixas intermediárias. Com esses intervalos definidos, a biblioteca *matplolib* é utilizada para montar o histograma: no *eixo x* ficam os valores possíveis de grau, e no *eixo y* o número de bairros que aparecem em cada um desses graus. Ao final, a imagem é salva em *out/histograma_graus.png*.



Por fim, a função *ranking_densidade_por_microrregiao()* utiliza a densidade da ego-rede de cada bairro, armazenada no arquivo *ego_bairro.csv*, e a associação entre microrregião e bairro, presente em *bairros_unique.csv*. A partir dessas duas fontes de dados, a função agrupa os bairros por microrregião e calcula a média das densidades dentro de cada grupo. Em seguida, é gerado um gráfico de barras horizontais que mostra cada microrregião com sua respectiva média, salvando a imagem em *out/ranking_densidade_microrregiao.png*. Essa visualização facilita enxergar quais áreas do Recife têm vizinhanças mais conectadas internamente (densidades maiores) e quais apresentam estruturas menos interligadas.



h. Apresentação interativa do grafo

Para facilitar a exploração visual da rede de bairros do Recife, foi desenvolvida uma representação interativa do grafo por meio da biblioteca pyvis, que permite a manipulação e navegação pela estrutura de forma dinâmica. A implementação está contida na função gerar_grafo_interativo(), localizada no arquivo viz.py, e é invocada através da função exploracoes_visuais() em solve.py.

A visualização interativa utiliza os dados consolidados nas etapas anteriores: o grafo principal (com todos os bairros e suas conexões), o dataframe de adjacências (com informações sobre as arestas), o dataframe de bairros (contendo as microrregiões) e o dataframe de ego-redes (com as métricas de densidade). A partir desses dados, a função cria uma instância da classe Network do pyvis, configurada com física habilitada para simular forças de atração e repulsão entre os nós, o que resulta em um layout orgânico e visualmente intuitivo.

Cada bairro é representado como um nó no grafo, sendo colorido de acordo com sua microrregião (RPAs 1.1 a 6.3), facilitando a identificação visual das diferentes áreas geográficas da cidade. Ao posicionar o cursor sobre um nó, são exibidas informações através de um tooltip customizado, que contém o nome do bairro, grau (número de conexões diretas), microrregião a que pertence e densidade da ego-rede. As arestas são representadas por linhas que conectam os bairros

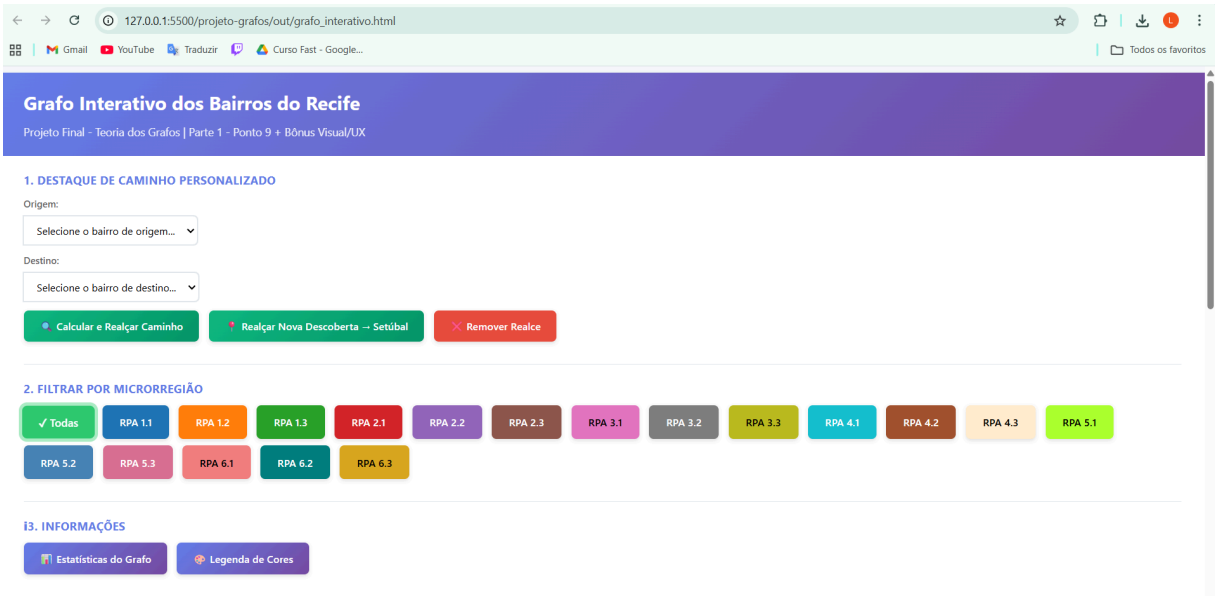
adjacentes, com espessura proporcional ao peso da conexão (definido pelo tipo de via: travessa 1.0, ponte 2.0, rua 3.0, avenida 4.0, BR 5.0), o que permite identificar visualmente as conexões mais importantes da rede.

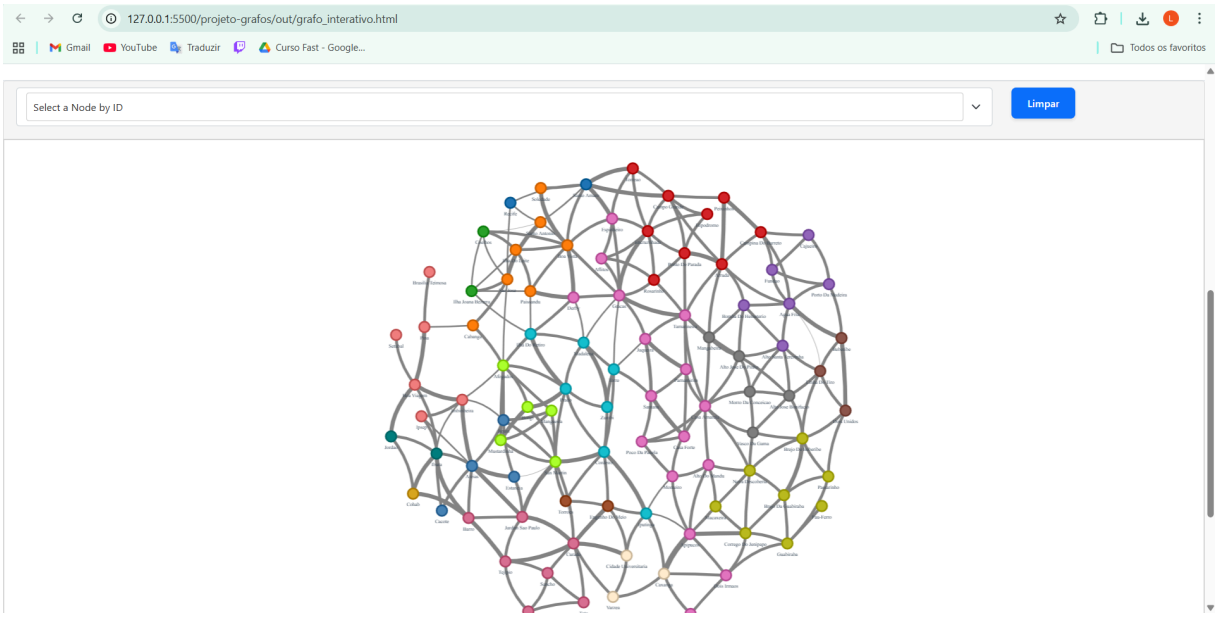
A interface interativa oferece recursos de navegação e busca: uma caixa de seleção permite localizar rapidamente qualquer bairro específico pelo nome, com a lista completa dos 94 bairros disponível em ordem alfabética. O controle de busca facilita a exploração direcionada sem necessidade de navegar manualmente pelo grafo. Além disso, é possível clicar e arrastar os nós para reorganizar a visualização, aproximar ou afastar (zoom com Ctrl+Scroll), e explorar as conexões de forma intuitiva.

Uma funcionalidade importante é o destaque do caminho obrigatório Nova Descoberta → Setúbal, acionado através de um botão dedicado na interface. Ao ser ativado, o caminho completo (Nova Descoberta → Alto do Mandu → Monteiro → Iputinga → Cordeiro → Prado → Afogados → Imbiribeira → Boa Viagem → Setúbal) é realçado em verde, permitindo visualizar claramente a sequência de bairros atravessados e sua posição relativa na estrutura global da rede.

A física do grafo foi ajustada para equilibrar legibilidade e estética, A força de repulsão entre os nós evita sobreposições excessivas, enquanto a atração das arestas mantém grupos conectados próximos uns dos outros. O resultado é uma visualização que reflete naturalmente a estrutura geográfica e topológica da cidade, onde bairros da mesma microrregião tendem a se agrupar visualmente por meio das cores atribuídas a cada região.

O arquivo gerado, grafo_interativo.html, pode ser aberto em qualquer navegador web moderno e não requer instalação de software adicional. A interatividade permite explorar a rede de forma intuitiva, identificar padrões de conectividade através das cores das microrregiões, visualizar o tipo de conexão pelo peso das arestas e compreender melhor as relações entre os diferentes bairros da cidade. Essa ferramenta é particularmente útil para análises exploratórias, apresentações e comunicação de resultados para públicos não técnicos, permitindo uma compreensão imediata e visual da estrutura urbana representada pelo grafo.





i. Bônus Visual e UX

Além dos recursos básicos de visualização, o grafo interativo implementa funcionalidades extras que enriquecem a análise e a exploração dos dados.

Cálculo de Caminhos com Dijkstra

Além da rota padrão "Nova Descoberta → Setúbal", a interface permite calcular e visualizar o caminho mais curto entre quaisquer dois bairros do Recife utilizando o algoritmo de Dijkstra. O usuário seleciona origem e destino em menus suspensos contendo os 94 bairros. Ao acionar "Calcular e Realçar Caminho", o algoritmo processa o grafo ponderado em tempo real. O trajeto é realçado visualmente e uma notificação exibe a rota completa e o número de saltos. O sistema também alerta caso não haja conexão, sendo uma ferramenta importante para planejamento urbano e análise de rotas.

Painel de Estatísticas Detalhadas

Expandindo as informações do *tooltip*, o clique em um nó abre um painel lateral fixo com dados completos: nome, grau (conexões), RPA, densidade da rede e lista de vizinhos (limitada aos 10 primeiros para melhor leitura, mas com indicação do número de vizinhos adicionais caso haja mais). O painel permite consultas prolongadas sem interferir na navegação, permanecendo visível até ser fechado manualmente ou substituído por nova seleção.

Filtros por Microrregião (RPA)

Botões coloridos na barra de controle permitem filtrar a visualização por RPA (1.1 a 6.3), mantendo a consistência das cores dos nós. Ao filtrar, os nós da região selecionada são ampliados e suas arestas destacadas em verde, enquanto os demais são reduzidos e tornados cinza. O botão "Todas" restaura a visão global. Isso permite focar nos padrões de conectividade interna de cada área geográfica.

Legenda e Estatísticas Globais

A interface oferece: 1) Legenda de Cores das microrregiões (RPAs), que abre um modal com as cores e contagens de bairros por RPA; e 2) Estatísticas do Grafo, que exibe métricas globais (total de nós/conexões) e reforça os saltos do caminho padrão (Nova Descoberta → Setúbal). Ambos aceleram a compreensão da estrutura dos dados sem necessidade de consultas externas.

3. Grafo de malhas aéreas

Para a segunda parte do trabalho, foi escolhido o *dataset* European Airlines Routes, disponível no Kaggle, por ser um conjunto de dados completo e bem estruturado sobre conexões entre aeroportos da Europa. Esse *dataset* inclui, dentre outras informações, as rotas no formato origem–destino e o preço de cada trecho, o que facilita a adaptação para um grafo dirigido e ponderado. Outra vantagem é que ele possui uma quantidade significativa de nós e arestas, permitindo testar o desempenho dos algoritmos em um cenário mais próximo do uso real em problemas de caminhos mínimos. Assim, a escolha desse conjunto de dados foi motivada pela clareza das informações, pela diversidade de rotas e pela compatibilidade direta com o tipo de análise exigida nesta etapa do trabalho.

a. Dijkstra

Para aplicar o algoritmo de Dijkstra, a implementação foi estendida para trabalhar com um novo conjunto de dados, agora representado pelas rotas aéreas entre aeroportos europeus: *europe_air_routes.csv*. Para tanto, após a finalização das análises sobre os bairros do Recife, o arquivo *cli.py* chama a função *executar_dijkstra_parte2()*, localizada em *solve.py*, que é responsável por conduzir toda a lógica dessa etapa.

O primeiro passo é construir o grafo de rotas aéreas, o que é feito pela função *construir_grafo_parte2()*, que por sua vez chama a função *carregar_dataset_parte2()* que carrega o *dataset*, identifica as colunas de origem e destino das rotas (códigos dos aeroportos) e a coluna de preço (pesos), remove linhas inválidas (com valores ausentes, strings vazias ou preços iguais a zero) e, a partir disso, gera uma lista de arestas no formato (origem, destino, peso).

Em seguida, essas arestas são inseridas em uma instância da classe Grafo, criada com o parâmetro *dirigido=True*, de modo que cada aeroporto passa a ser representado como um nó e cada voo disponível se torna uma aresta dirigida, ponderada pelo valor da passagem. Com o grafo de rotas aéreas construído, a função *executar_dijkstra_parte2()* passa então à etapa de definição das consultas de caminhos mínimos.

Para tanto, em vez de escolher pares de aeroportos automaticamente, foi criado manualmente o arquivo *pares_parte2.csv*, contendo uma lista de combinações origem–destino que são de interesse para análise, a exemplo do trecho LIS - CDG. Esse arquivo é lido dentro da própria função, que valida se as colunas origem e destino estão presentes e, a partir delas, monta uma lista de pares a serem processados.

Para cada par de aeroportos listado em *pares_parte2.csv*, a função aplica o algoritmo de Dijkstra sobre o grafo de rotas. Isso é feito chamando duas funções utilizadas na Parte 1: *dijkstra_path()*, que devolve a sequência de nós que corresponde ao caminho de menor custo entre origem e destino, e *dijkstra_path_length()*, que retorna o valor total desse custo, considerando a soma dos preços das arestas percorridas.

Antes de iniciar cada cálculo, é feita uma marcação de tempo com *time.perf_counter()*, o que permite medir o tempo de execução do algoritmo para aquele par específico; ao final, o tempo decorrido é armazenado junto com o resultado do caminho. Cada consulta gera um registro contendo a origem, o destino, o custo mínimo encontrado, a descrição textual do percurso e um status indicando se a busca foi bem-sucedida ou se ocorreu algum erro (por exemplo, ausência de caminho no grafo).

Por fim, os resultados são consolidados: os dados dos caminhos mínimos são organizados em um *dataframe* e salvo em *parte2_dijkstra.json*. Em seguida, é montado também um arquivo de relatório (*parte2_report.json*), que registra informações globais do grafo utilizado (como o número de nós e arestas) e a lista de tarefas executadas, incluindo o tempo de processamento para cada par origem–destino.

b. BFS

O algoritmo BFS (Breadth-First Search) também foi implementado para explorar o grafo de rotas aéreas europeias de forma sistemática, visitando todos os aeroportos alcançáveis a partir de um ponto de origem, nível por nível. A implementação está presente no arquivo algorithms.py e é invocada pela função *executar_bfs_parte2()*, no arquivo *solve.py*.

Para garantir uma análise diversificada da rede, foram selecionados: o primeiro aeroporto da lista (TZL), um aeroporto intermediário (HOV) e o último aeroporto (ACH). Essa escolha permitiu observar diferentes padrões de conectividade dentro do grafo. O algoritmo utiliza uma fila (implementada com o deque do Python) para processar os nós em ordem de descoberta. Para cada aeroporto de origem, o BFS realiza as seguintes operações: marca o nó inicial como visitado e o coloca no nível 0; em seguida, explora todos os vizinhos diretos (aeroportos com voos diretos), marcando-os como visitados e atribuindo-os ao nível 1; o processo continua iterativamente, expandindo para os vizinhos dos vizinhos, e assim sucessivamente, até que não haja mais nós alcançáveis. A função retorna um dicionário contendo: o conjunto de nós visitados, os níveis (ou camadas) de cada nó em relação à origem, a estrutura de parentesco (que permite reconstruir caminhos) e a ordem de visitação.

Os resultados mostraram comportamentos distintos para cada origem testada. A partir de TZL, o algoritmo conseguiu alcançar todos os 810 aeroportos do grafo, distribuídos em 5 níveis, indicando uma forte conectividade desse hub com o restante da rede europeia. Por outro lado, os aeroportos HOV e ACH apresentaram conectividade limitada, sugerindo que são aeroportos isolados ou com conectividade restrita no dataset utilizado, é precisamente o que ocorre: de acordo com o dataset, ambas têm apenas voos chegando. Todos os resultados, incluindo o número de nós visitados, a distribuição por níveis e o tempo de execução de cada busca, foram armazenados no arquivo *parte2_bfs.json*. O tempo de processamento foi medido utilizando time.perf_counter(), permitindo a comparação de desempenho com os demais algoritmos implementados.

c. DFS

A implementação do algoritmo DFS (Depth-First Search) permite explorar o grafo de rotas aéreas em profundidade, visitando um caminho completo antes de retroceder para explorar outras alternativas. Diferente do BFS, que expande horizontalmente por níveis, o DFS avança verticalmente, seguindo cada rota até seu limite antes de retornar.

Sua implementação é encontrada no arquivo algorithms.py e é também executada pela função *executar_dfs_parte2()* em *solve.py*. O DFS foi aplicado aos mesmos três aeroportos utilizados no BFS (TZL, HOV e ACH), garantindo comparabilidade entre os resultados. O algoritmo utiliza recursão para visitar os nós, mantendo informações sobre: o conjunto de nós visitados, a estrutura de parentesco, a ordem de visitação, os tempos de descoberta e finalização de cada nó, a detecção de ciclos e a classificação das arestas.

Os resultados obtidos revelaram padrões interessantes sobre a estrutura da rede aérea europeia. A partir de TZL, o DFS visitou todos os 810 aeroportos e detectou a presença de ciclos no grafo (has_cycle=True), o que era esperado em uma rede de rotas aéreas onde existem voos de ida e volta. A classificação das arestas mostrou 809 arestas de árvore, 6.384 arestas de retorno, 8.663 arestas de avanço e 292 arestas de cruzamento, evidenciando a complexidade e densidade das conexões na malha aérea europeia. Para os aeroportos HOV e ACH, assim como observado no BFS, o DFS visitou apenas o próprio nó de origem, confirmando o isolamento desses aeroportos no dataset.

O tempo de execução do DFS foi consistentemente maior que o do BFS devido à natureza recursiva do algoritmo e ao overhead de manter múltiplas informações adicionais (tempos de descoberta, finalização e classificação de arestas). Todos os resultados, incluindo estatísticas de visitação, detecção de ciclos, classificação de arestas e métricas de tempo, foram salvos em *parte2_dfs.json*.

d. Bellman-Ford

O algoritmo de Bellman-Ford foi implementado para calcular caminhos mínimos em grafos que podem conter arestas com pesos negativos, uma situação em que o algoritmo de Dijkstra não é aplicável.

O código está presente em algorithms.py e é executado pela função *executar_bellman_ford_parte2()* em *solve.py*. A implementação segue o procedimento clássico do algoritmo: primeiro, inicializa as distâncias de todos os nós como infinito, exceto o nó de origem que recebe distância zero; em seguida, realiza |V|-1 iterações (onde |V| é o número de vértices), relaxando todas as arestas em cada iteração; por fim, executa uma iteração adicional para detectar ciclos negativos, verificando se alguma aresta ainda pode ser relaxada.

O primeiro cenário utiliza o grafo real de rotas aéreas com pesos positivos, aplicando o Bellman-Ford a três pares de aeroportos (LIS→CDG, MAD→FRA e BCN→AMS). Os resultados confirmaram que o algoritmo encontra os mesmos caminhos mínimos que o Dijkstra quando não há pesos negativos, validando a implementação também dos anteriores. Por exemplo, o custo de LIS para CDG foi calculado como 50.0, idêntico ao resultado obtido pelo Dijkstra.

O segundo cenário foi criado sinteticamente para testar o comportamento do algoritmo com pesos negativos. Foi construído um pequeno grafo onde algumas arestas receberam pesos negativos controlados, garantindo que o custo total de qualquer ciclo permanecesse positivo. O algoritmo processa corretamente essas arestas negativas, calculando as distâncias mínimas sem detectar ciclos negativos, demonstrando sua capacidade de lidar com essa característica que distingue o Bellman-Ford do Dijkstra.

O terceiro cenário testou a detecção de ciclos negativos, requisito fundamental para validar a implementação. Foi criado um grafo sintético com três nós (X, Y, Z) formando um ciclo onde a soma dos pesos é negativa: X→Y (peso 1.0), Y→Z (peso 1.0) e Z→X (peso -5.0), totalizando -3.0. O algoritmo detectou corretamente esse ciclo negativo e retornou a sequência de nós que o compõe (['Y', 'X', 'Z']), confirmando que a implementação atende ao requisito de identificação desse tipo de estrutura.

Em termos de desempenho, o Bellman-Ford apresentou tempo de execução significativamente maior que o Dijkstra, conforme esperado. Enquanto o Dijkstra tem

complexidade O((V+E)log V) usando heap, o Bellman-Ford possui complexidade O(V×E), tornando-o mais lento especialmente em grafos densos como a malha aérea europeia (810 nós e 16.148 arestas).

Todos os resultados dos três cenários testados, incluindo custos calculados, detecção de ciclos negativos, caminhos encontrados e métricas de tempo de execução, foram consolidados e salvos no arquivo *parte2_bellman_ford.json*.

e. Discussão Crítica: Comparação dos Algoritmos

A implementação e execução dos quatro algoritmos de busca em grafos sobre o dataset de rotas aéreas europeias permitiu observar características distintas que determinam a adequação de cada um para diferentes cenários de aplicação.

BFS é o algoritmo mais adequado quando o objetivo é encontrar o caminho com o menor número de saltos entre dois pontos, ignorando os pesos das arestas. No contexto de rotas aéreas, isso corresponderia a encontrar o trajeto com o menor número de escalas, sem levar em conta os custos dos voos. O BFS também é útil para análises de alcançabilidade, como foi observado na detecção dos aeroportos isolados (HOV e ACH). Sua complexidade temporal de O(V+E) o torna eficiente para grafos esparsos, e a execução a partir de TZL demonstrou tempos de processamento extremamente baixos (0,0007 segundos), mesmo visitando todos os 810 aeroportos.

DFS destaca-se quando é necessário explorar toda a estrutura do grafo de forma sistemática, detectar ciclos ou classificar arestas. No projeto, a aplicação do DFS revelou informações estruturais importantes sobre a rede aérea: a presença de 6.384 arestas de retorno confirma que existem ciclos no grafo (voos de ida e volta), as 8.663 arestas de avanço indicam conexões que "pulam" níveis na árvore de busca, e as 292 arestas de cruzamento mostram rotas entre ramos diferentes da rede. Essa classificação é valiosa para entender padrões de conectividade que não são evidentes em outros algoritmos. Entretanto, o DFS não é apropriado para encontrar caminhos mínimos ponderados, pois não considera os pesos das arestas em sua exploração.

Dijkstra é o algoritmo de escolha quando se busca o caminho de menor custo em grafos com pesos não-negativos. No contexto de rotas aéreas com preços sempre positivos, o Dijkstra forneceu a solução ótima com eficiência superior ao Bellman-Ford. Utilizando uma fila de prioridade (heap), permite processar o grafo de 810 nós e 16.148 arestas em tempos aceitáveis (entre 0,008 e 0,041 segundos por consulta). A implementação demonstrou que, para os três pares testados com pesos positivos, o Dijkstra e o Bellman-Ford encontraram exatamente os mesmos custos (50.0, 90.0 e 60.0), confirmando a corretude de ambos. A limitação fundamental do Dijkstra é sua incapacidade de lidar com pesos negativos.

Bellman-Ford é o único algoritmo entre os quatro capaz de trabalhar corretamente com pesos negativos e detectar ciclos negativos. Essa capacidade foi demonstrada nos cenários sintéticos criados: no teste com pesos negativos sem ciclo, o algoritmo calculou corretamente o custo de 4.0 de A para E, utilizando arestas com pesos -8 e -3; no teste com ciclo negativo, detectou corretamente o ciclo formado por X→Y→Z→X com custo total de -3. Entretanto, essa versatilidade tem um custo computacional: com complexidade O(V×E), o Bellman-Ford é significativamente mais lento que o Dijkstra. Para os mesmos pares de aeroportos, foram aproximadamente 200 a 600 vezes maiores (0,004 segundos vs 0,008-0,041 segundos). Portanto, o Bellman-Ford só deve ser utilizado quando o grafo contém ou pode conter pesos negativos; caso contrário, o Dijkstra é preferível por questões de desempenho.

É importante ressaltar os limites do design de pesos: o projeto adotou o preço das passagens como peso das arestas no grafo de rotas aéreas (coluna price no dataset), uma escolha natural e intuitiva que, no entanto, apresenta limitações importantes que devem ser reconhecidas. Os preços de passagens aéreas são altamente voláteis e dependem de múltiplos fatores não capturados pelo modelo (época do ano, antecedência da compra, disponibilidade de assentos, promoções, horário do voo, classe de serviço, entre outros).

REFERÊNCIAS

GOOGLE. Mapas dos bairros da cidade do Recife. Disponível em: https://www.google.com/maps/place/Recife,+State+of+Pernambuco. Acesso em: 05 nov. 2025.

KAGGLE. European Airlines Routes. Disponível em: https://www.kaggle.com/datasets/lunthu/european-airlines-routes. Acesso em: 13 nov. 2025.

NETWORKX. Shortest path algorithms for weighted graphs. Disponível em: https://networkx.org/documentation/stable/_modules/networkx/algorithms/shortest_paths/weighted.html#dijkstra_path. Acesso em: 09 nov. 2025.

PREFEITURA DA CIDADE DO RECIFE. Perfil dos Bairros. Disponível em: https://www2.recife.pe.gov.br/servico/perfil-dos-bairros. Acesso em: 08 nov. 2025.