Grupo 6 - 5A

- Felipe França
- Felipe Matias
- Lucas Ferreira Torres
- Luis Gustavo Melo

Estrutura de Arquivos

```
projeto-grafos/
├─ README.md
├─ requirements.txt
├─ data/
│  ├─ bairros_recife.csv
│  ├─ adjacencias_bairros.csv
│  ├─ enderecos.csv
│  └─ dataset_parte2/
│     ├─ airlineFlightRoutes.csv
│     └─ csvFiltrado.csv
├─ out/
│  └─ .gitkeep
├─ src/
│  ├─ cli.py — Interface de Linha de Comando
│  ├─ solve.py — Todas as funções que geram resultados .csv ou .json
│  ├─ graphs/
│  │  ├─ io.py — Derretimento do Dataset Original da Parte 1
│  │  ├─ graph.py — Geração da Lista de Adjacência
│  │  └─ algorithms.py — Dijkstra, Bellman-Ford, DFS, BFS
│  └─ viz.py — Todas as funções que geram resultados .png ou .html
├─ tests/
│  ├─ test_bfs.py
│  ├─ test_dfs.py
│  ├─ test_dijkstra.py
│  └─ test_bellman_ford.py
└─ relatorio.pdf
```

Como Executar

Para gerar todos os arquivos do out/, você deve realizar os seguintes passos

1. Criar o ambiente virtual

```
py -m venv venv
```

2. Ativar o ambiente virtual

```
.\venv\Scripts\activate
```

3. Instalar requisitos

```
pip install -r requirements.txt
```

4. Gerar resultados em /out (execute na pasta raiz do projeto, ou seja, ./projeto-grafos)

```
python -m src.cli out
```

PS: Você também pode rodar os arquivos solve.py e viz.py manualmente para gerar todos os arquivos de /out.

Parte 1

1) Processamento do dataset original

1.1) (io.py)

Objetivo:

- “Derreter” o dataset
- Gerar uma lista única de bairros
- Associar cada bairro à sua microrregião

Função derreter_bairros_recife()
Responsável por construir um dicionário {bairro -> microrregião}

Função salvar_bairros_unique()
Depois de derreter, salvamos um CSV:

Bairro, microrregião
Boa viagem, 6.1
Pina, 6.1
…

O csv bairros_unique é usado para:

- Construir subgrafos por microrregião
- Calcular métricas (Ordem, Tamanho, Densidade)
- Análises de grau
- Ego-networks

1.2) Construção do grafo dos bairros (Lista de adjacências)

O módulo graph.py é responsável por transformar esse CSV em uma lista de adjacências, que é a estrutura básica para rodar BFS/DFS/Dijkstra

Função carregar_lista_adjacencia()

- Lê o csv da lista de arestas adjacencias_bairros.csv
- Normaliza o nome dos bairros
- Converte o peso para float
- Inserte no grafo

O resultado é um dicionário com a lista de adjacências

2) Métricas de definição dos Logradouros/Arestas

As arestas são definidas como o caminho entre os centros dos bairros e o logradouro é a via que os conecta. Esse caminho é definido pelo google maps. Ao selecionar um bairro como origem e outro como destino, obtém-se opções de rota. O caminho selecionado é o que possui a menor distância. Em caso de distâncias semelhantes, a primeira opção é escolhida.

O logradouro é definido através das ruas que conectam as fronteiras entre os bairros. Caso a rota mais curta de mais de dois ou mais bairros cruzem fronteiras por um mesmo logradouro, uma nova rota é selecionada manualmente no google maps. Esse tratamento se faz necessário para evitar a criação de hiper arestas com o mínimo de interferência na distância.

Para obter as fronteiras entre os bairros, foi utilizado como referência o site da Prefeitura do Recife.

3) Métrica Globais e por Grupo

As funções responsáveis por calcular as métricas globais e pro grupo, e de ego-network podem ser encontradas no arquivo solve.py. Sendo elas:

3.1) def metricas_globais(lista_adjacencia, write)

Calcula as métricas globais de todos o recife, retorna o resultado e opcionalmente gera o arquivo de resultado ***out/recife_global.json***.

3.2) def obter_subgrafo_por_microrregiao(lista_adjacencia, df, microrregiao)

A partir da lista de adjacência, bairros_unique.csv, e o número da microregião (ex: 1.1, 5.2, etc), gera uma lista de adjacência do subgrafo.

3.3) def metricas_globais_microrregioes(lista_adjacencia)

Utilizando da função anterior para obter a lista de adjacência da microregião, utiliza também da primeira função para gerar um json com os resultados da tal microregião, então armazena todos os resultados em um arquivo ***out/microrregioes.json***

3.4) def ego_network_metricas(lista_adjacencia)

Obtem a ego network de cada bairro e calcula suas métricas globais, assim como seu grau. Os resultados são armazenados em um arquivo ***out/ego_bairro.csv***

4) Graus e rankings

4.1) graus.csv

No arquivo solve.py, existe a função def gerar_csv_graus(lista_adjacencia), ela é responsável por gerar o CSV com os graus de cada bairro.

4.2) Bairro mais denso

O ranking é encontrado no arquivo ***out/ranking_densidade_ego_por_microrregiao.png****,* essa imagem é gerada no arquivo viz.py, que será explicado na seção 8 deste documento.

4.3) Bairro com maior grau

No arquivo solve.py, existe a função def obter_bairro_com_maior_grau(), ela obtem o bairro com maior grau do CSV gerado no ponto 4.1, e gera o arquivo ***out/bairro_maior_grau.json*** com as informações.

5) Métricas de Peso das Arestas

O peso das arestas é representado pela distância em quilômetros entre os centros dos bairros que a aresta conecta. Essa distância é calculada através do caminho mais curto à pé calculado pelo google maps. A escolha dessa métrica diminui variabilidade e torna mais consistente os valores, pois, quando se desloca à pé, variáveis atreladas à veículos motorizados como engarrafamentos, horários de pico, sentido das vias e sinais são desconsideradas.

Em contrapartida, o sistema de pesos torna-se muito específico para percursos à pé ou, pelo menos, independentes das variáveis de veículos motorizados.

6) Distância entre endereços X e Y

Foi escolhido 5 pares de endereços, dentre eles “Nova Descoberta → Setúbal” e armazenados no arquivo data/enderecos.csv. Para encontrar o caminho e o custo entre os dois endereços é utilizada a função def calcular_peso_caminho_enderecos(lista_adjacencia) encontrada no solve.py. Como resultado, a função gera dois arquivos:

1. out/distancias_enderecos.csv: Contem o resultado da distancia entre todos os pares de endereços no csv.
2. out/percurso_nova_descoberta_setubal.json: Contem o resultado especificamente entre Nova Descoberta → Setúbal.

7) Visualizações (viz.py)

Essa seção vai se referir ao ponto 7, 8 e 9 das especificações do projeto final.

Todos os arquivos mencionados nessa seção são gerados por funções no arquivo viz.py

O arquivo principal de visualização da parte 1 é o ***out/grafo_interativo.html***, que contem o grafo interativo dos bairros de recife, uma funcionalidade de traçar e calcular peso de qualquer caminho escolhido, e um botão que destaca o caminho Nova Descoberta → Setúbal automaticamente. Além disso, os nós e arestas possuem tooltips com suas informações específicas, como métricas e peso e a grossura das arestas representa o seu peso. O grafo interativo é gerado pela funçao def visualizar_grafo.

Dentro da visualização, existe um botão para alternar para a visualização do digrafo da parte 2.

Também existe um arquivo separado para visualizar somente a árvore do caminho Nova Descoberta → Setúbal, pode ser encontrando em out/arvore_percurso.html. É gerado poela função def plot_percurso_nova_descoberta_setubal().

Sobre as 3 visualizações adicionais requisitadas:

1. Mapa de Cores por Grau: Encontrada dentro do grafo interativo, e explicada na legenda da página. Existe outro arquivo alternativo separado para essa visualização tambem, sendo ele ***out/mapa_cores_graus.html.*** Essa visualização é gerada pela função def ranking_densidade_ego_por_microrregiao. Ao analisar essa visualização, pode-se entender quais são os bairros mais conexos e importantes do Recife, como o bairro das Graças, que possui o maior grau.

2. Ranking de Densidade Ego por Microrregião: Encontrada em ***out/ranking_densidade_ego_por_microrregiao.png***, um gráfico de barras

demonstrando o ranking das principais microrregiões, onde o X é a densidade ego e o Y seriam as microrregiões. Essa visualização é gerada pela função def mapa_de_cores_por_grau. Ao analisar essa visualização, pode-se entender quais são as microrregiões com a maior densidade, ou seja, com os bairros mais conectados entre si.

3. Histograma de Graus: Encontrado em **out/histograma_graus.png**, um histograma demonstrando a frequência do valor de graus dos bairros. Essa visualização é gerada pela função def histograma_graus. Ao analisar essa visualização, pode-se entender a frequência do número de graus dos bairros, é possível logo de cara ver que a maior parte dos bairros possui um grau igual a 6.

4. Distribuição de Graus: Encontrado em **out/distribuicao_graus.png**, um gráfico demonstrando a frequência dos aeroportos e os graus de saída deles(voos).A análise dele se encontra na Parte 2, ponto 4.1 deste relatório

Parte 2

1) Dataset Escolhido

O dataset escolhido para a parte 2 foi um recorte filtrado do arquivo Consumer Airline report, que contém informações sobre rotas aéreas entre aeroportos dos Estados Unidos. Para adequação ao tamanho recomendado, foi realizado um filtro mantendo apenas as rotas do ano de 2024.

Após a filtragem, o dataset resultante obteve:

- 136 vértices
- 1905 arestas
- Tipo: Dirigido e Ponderado

Os dados foram convertidos em uma lista de adjacência no formato:

Aeroporto_origem -> [(destino, peso), (destino2, peso2),...]

A partir da lista de adjacência, foram calculados:

- Grau de saída: quantidade de voos que partem do aeroporto
- Grau de entrada: quantidade de voos que chegam ao aeroporto

2) Execução dos Algoritmos

2.1) BFS e DFS

Foram executados BFS e DFS para os vértices:

ABQ, ACY e COS

Os resultados foram salvos em: out/bfs_dfs_resultados.json

2.2) Dijkstra

Como todos os pesos do dataset são distância em milhas (sempre >= 0), Dijktra pode ser aplicado sem restrições.

Os pares usados:

- **dfw -> mia**
- **lax -> ord**
- **bos -> sea**
- **phx -> den**
- **atl -> iah**

Para cada par foram armazenados:

- Custo total (Distância mínima)
- Caminho obtido (Lista de aeroportos)
- Casos impossíveis retornam “caminho”: “inexistente”

Os resultados foram salvos em: out/dijkstra_resultados.json

2.3) Bellman-Ford

Conforme exigido, foram criados dois cenários artificiais usando códigos reais de aeroportos, mas com pesos definidos manualmente para testar o algoritmo.

Caso 1 - Com peso negativo, mas sem ciclo negativo

- dfw -> mia (120)
- mia -> ord (-20)
- dfw -> ord (50)

Resultado esperado: distâncias mínimas corretas.

- dfw : 0
- mia: 120
- ord: 50

Caso 2 - Com ciclo negativo

- lax -> phx (3)
- phx -> sea (-10)
- Sea -> lax(2) # ciclo: lax -> phx -> sea -> lax (peso negativo)

Resultado esperado: O algoritmo detecta o ciclo negativo e retorna -1.

Os resultados foram salvos em: out/bellman_ford_resultados.json

3) Métricas de desempenho

Para atender ao requisito de métricas de desempenho, desenvolvemos a função **executar_metrica_desempenho** dentro do script **solve.py**. O objetivo foi cronometrar a execução de cada algoritmo e exportar esses dados para a tabela **out/parte2_report.json**(Saída obrigatória requerida).Para garantir a precisão das medições de tempo, optamos por utilizar a função **time.perf_counter()**, que é mais exata para intervalos de tempo curtos do que o cronômetro padrão.

Como o dataset de voos é muito grande, rodar os algoritmos em todos os vértices seria demorado e ineficiente. Por isso, configuramos o código para executar o **BFS**,**DFS** e o **Bellman Ford** apenas em 3 fontes de origem distintas, com o Bellman Ford tendo os casos de controle também, e o Dijkstra com 5 pares de origem-destino pré definidos, o que é suficiente para validar o funcionamento e obter uma média de tempo confiável.

Os resultados foram salvos em: out/parte2_report.json e gerados em solve.py

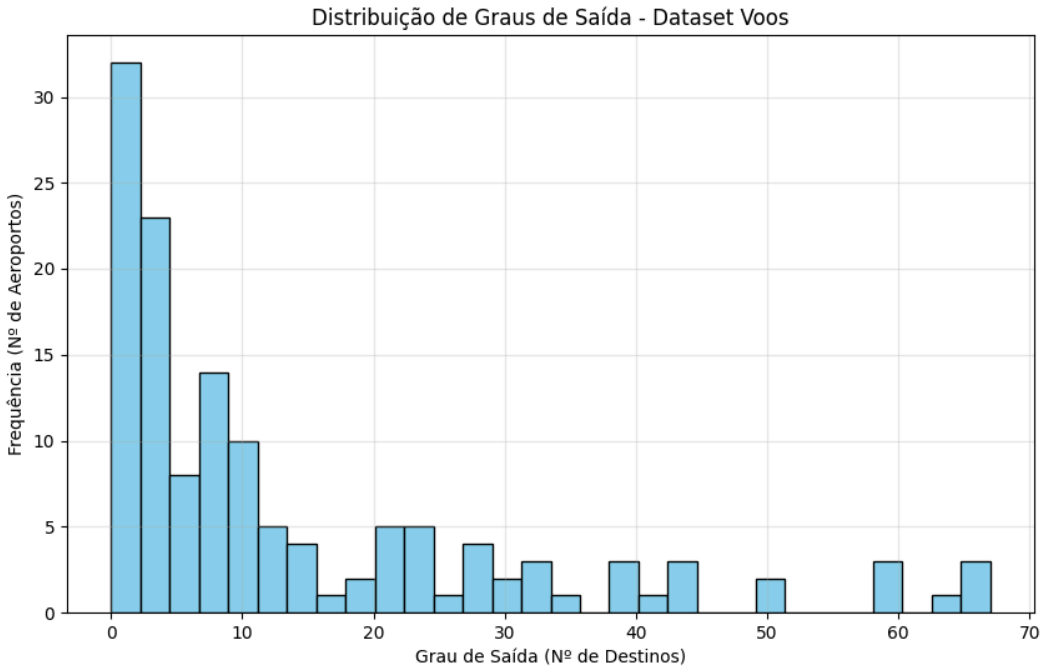
- BFS teve uma média de desempenho de 0.00005s
- DFS teve uma média de desempenho de 0.04s
- Dijkstra teve uma média de desempenho de 0.0001s
- Bellman Ford teve uma média de desempenho de 0.016s

Discussões acerca dos algoritmos segue na parte 5.

4) Visualização

4.1) Distribuição de Graus

Para poder visualizar melhor o grafos, desenvolvemos um gráfico de distribuição de graus, gerado pela função **`gerar_grafico_distribuicao_graus`**, implementada no módulo de visualização,processando a lista de adjacência para extrair a contagem de destinos diretos de cada aeroporto e plotar esses dados em um histograma.

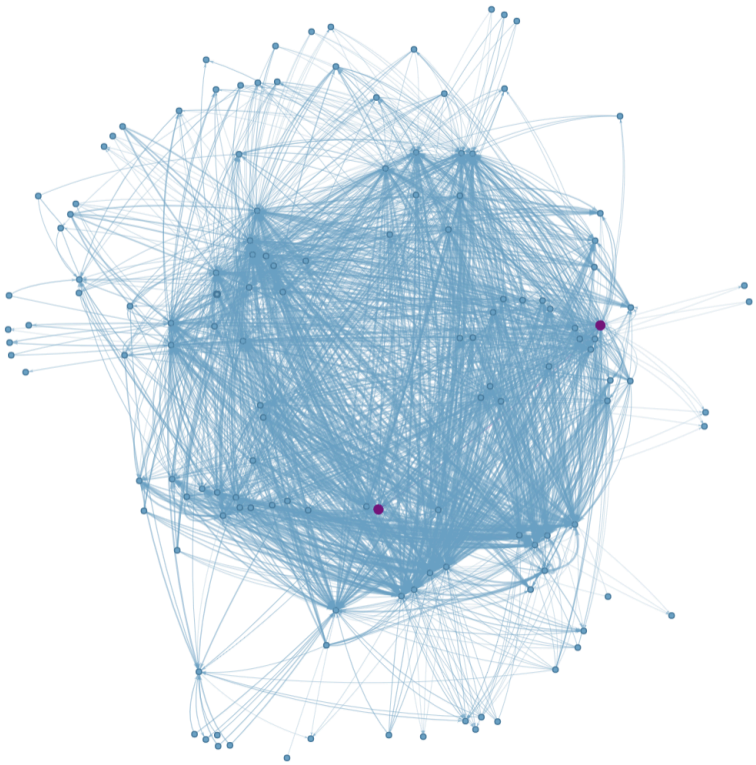


Verificando e analisando o gráfico da para ver que a maioria dos aeroportos tem um grau de saída baixo(entre 1 e 10), sendo provavelmente aeroportos regionais ou locais, longe dos aeroportos de grandes centros urbanos, que contém bem mais graus de saída e são poucos, mas concentram a maior parte das conexões entre os voos.

Os resultados foram salvos em: (**`out/distribuicao_graus.png`**) e gerados em **(src\viz.py)**

4.2) Digrafo

A visualização do digrafo pode ser encontrada em **(out/digrafo_visualizacao.html)** e gerados em **(src\viz.py)** Além disso, existe um botão que redireciona o usuário para a visualização da parte 1.



Origem: dal Destino: hou Menor Caminho (Bellman-Ford) **Custo: 247**

Trocar Grafo: Bairros de Recife

Verificando e analisando o grafo de voos acima, mostrando os aeroportos(vértice) e as rotas de voos(arestas) da pra visualizar melhor ainda que o gráfico de distribuição de graus como existem poucos aeroportos com a maior parte das conexões, sendo uma rede bem densa e carregada de conexões em poucos vértices.

5. Discussão Crítica

Nesta parte discutiremos quando e porque cada algoritmo é mais adequado, e os limites dos nossos design de pesos. Utilizando como base os testes de desempenho realizados e salvo em (***out/parte2_report.json***).

5.1. BFS (Busca em Largura):

- **Desempenho:** Foi o algoritmo mais rápido do projeto 0.00005s.
- **Por que:** Implementação simples e utilização de filas (queue) como estrutura de dados usada nas iterações. Verificação de visitação de vértices a partir de um dicionário. Sua complexidade de tempo sendo $O(V + E)$
- **Quando é adequado:** É a escolha ideal para encontrar o caminho com o **menor número de arestas** (conexões dos voos), independentemente da distância. No contexto de voos, o BFS responderia "qual a rota com menos conexões", mas ignoraria a distância total em milhas.

5.2.DFS (Busca em Profundidade):

- **Desempenho:** Consideravelmente mais lento que o BFS 0.04s.
- **Por que:**Implementação mais complexa que BFS por causa da característica de recursão (profundidade = recursão),gastando sempre um tempo a mais por conta de ficar chamando a função auxiliar.Além disso, o DFS tende a explorar ramos longos desnecessários antes de chegar no destino. Sua complexidade de tempo sendo $O(V + E)$
- **Quando é adequado:** Bom, ele não é recomendado para buscar rotas curtas (caminho mínimo). Sua principal utilidade neste projeto foi permitir a detecção de ciclos e componentes através da classificação das arestas de retorno.

5.3.Dijkstra (Caminho Mínimo Ponderado):

- **Desempenho:** Muito eficiente 0.0001s.
- **Por que:** Utilização de fila de prioridade (min-heap) para obter eficiência. Detecção de pesos negativos. Sua complexidade de tempo é $O((V+E)*logV)$
- **Quando é adequado:** Ele geralmente é um dos melhores para sistemas de navegação reais(GPS,rotas aéreas) onde os pesos não são negativos, garantindo o menor custo total bem mais rápido que o Bellman-Ford.

5.4.Bellman-Ford:

- **Desempenho:** O mais lento 0.016s, sendo cerca de **160 vezes mais lento** que o Dijkstra nos nossos testes.
- **Por que:** Sua complexidade O(V*E) obriga o algoritmo a "relaxar" todas as arestas do grafo repetidas vezes, o que é ineficiente para grafos densos ou grandes, como no nosso caso dos voos.
- **Quando é adequado:** Ele é adequado em cenários específicos onde existem **pesos negativos** (ex: transações financeiras com lucro ou jogos), pois é o único capaz de calcular rotas corretas nessas condições e detectar ciclos negativos (como demonstrado nos nossos casos de controle).

5.5. Limites do Design de Pesos

A modelagem dos design de pesos adotada impõe certas fronteiras à análise:

1. **Unidimensionalidade:** Ao considerar apenas a distância, o design de pesos ignora fatores críticos para o mundo real, como tempo de voo, preço da passagem ou tempo de espera em conexões. O caminho "mais curto" em milhas ,nem sempre é o melhor para o usuário.

   Na parte 1 também temos o mesmo problema, ignorar fatores essenciais do mundo real como trânsito,custo, clima e condições climáticas,desvios,obras, mesmo calculando o melhor caminho, ele na vida real pode muito bem não ser verdadeiramente o melhor caminho.

2. **Inexistência de Pesos Negativos:** O design baseado nas distâncias impede a existência de pesos negativos, o que na vida real seria interpretado como ganhar “tempo”, no caso de voos teria a questão dos fusos horários onde poderia ganhar esse tempo.

6. Testes

Foram feitos testes automatizados utilizando o pytest. Para rodar os teste é só digitar no terminal:

```
\projeto-grafos> python -m pytest
```

python -m pytest

6.1 test_bfs

Objetivo: Testar os níveis corretos em um grafo pequeno.

- **Cenário de Teste (`test_bfs_basico`):**Fizemos um grafo simples onde um nó raiz **A** conecta-se a dois filhos **B** e **C**.
- **Validação:** O teste verifica se, após visitar a raiz A, o algoritmo visita obrigatoriamente os nós B e C (nível 1) antes de qualquer profundidade adicional. Isso confirma que a fila (*queue*) está funcionando corretamente, respeitando a ordem de visitação por distância em arestas.

6.2 test_dfs

Objetivo: Testar a detecção de ciclo e classificação de arestas.

- **Teste de Classificação (test_dfs_classificacao_arestas):** Em um grafo linear ABC, verifica-se se o algoritmo segue a profundidade máxima antes de retornar, confirmando o comportamento de Aresta de Árvore.
- **Teste de Ciclo (test_dfs_deteccao_ciclo):** Simula-se um grafo com ciclo A para B e B para A. A aresta de volta B para A caracteriza uma Aresta de Retorno. O teste valida se o algoritmo identifica que o nó A já é um ancestral na pilha de recursão e interrompe a execução, prevenindo *loops* infinitos e retornando a lista correta de visitados.

6.3 test_djisktra

Objetivo: Testar os caminhos corretos com pesos ≥ 0; recusar dado com peso negativo.

- **Teste de Otimização (test_dijkstra_caminho_correto):** Fizemos um caminho direto de A para B possuindo um custo alto (10) e o caminho indireto de A para C e para B possuindo um custo baixo (2). O teste confirma que o algoritmo ignora o caminho teoricamente mais curto em favor do caminho com menor peso total, validando a lógica do *Min-Heap*.
- **Teste de Segurança (test_dijkstra_rejeita_pesos_negativos):** Como o algoritmo de Dijkstra não garante corretude com pesos negativos, implementamos uma verificação de segurança. O teste insere uma aresta com peso -5 e confirma se a função retorna a *flag* de erro (-1), garantindo que o sistema não produza resultados falsos.

6.4 test_bellman_ford

Objetivo: Testar o algoritmo (i) com pesos negativos sem ciclo negativo → distâncias corretas;

(ii) com ciclo negativo → flag.

- **Teste de Peso Negativo (test_bf_com_pesos_negativos):** Utiliza-se um grafo onde um "atalho" possui peso negativo, reduzindo o custo total do caminho. O teste confirma que o algoritmo calcula a distância correta (10 - 5 = 5), provando que ele realiza o relaxamento completo das arestas.
- **Teste de Ciclo Negativo (test_bf_ciclo_negativo):** Simula-se um loop com saldo negativo (soma das arestas < 0). O teste verifica se a "iteração extra" do algoritmo detecta a possibilidade de redução infinita de custo e retorna a *flag* de erro (-1) que indica um ciclo negativo no nosso algoritmo.

Referências

G-32. Dijkstra's Algorithm - Using Priority Queue - C++ and Java - Part 1
G-41. Bellman Ford Algorithm
G-5. Breadth-First Search (BFS) | C++ and Java | Traversal Technique in Graphs
G-6. Depth-First Search (DFS) | C++ and Java | Traversal Technique in Graphs
G-35. Print Shortest Path - Dijkstra's Algorithm
Site da Prefeitura do Recife - Bairros do Recife
Geeks for Geeks - Dijkstra's Algorithm
Geeks for Geeks - Bellman–Ford Algorithm
Geeks for Geeks - BFS
Geeks for Geeks - DFS