

PROJETO DE GRAFOS

Davi Gomes, Heloísa Tanaka, Isabela Spinelli, João
Pedro Fontes, Maria Luísa Arruda



DERRETIMENTO

- NFKD (Formulário de Normalização de Compatibilidade e Decomposição)
- normalize (Remove caracteres especiais)
- slug (Cria strings seguras para nomear arquivos de saída)

```
● ● ●  
1  def _normalize(nome: str) -> str:  
2      if nome is None:  
3          return ""  
4      t = unicodedata.normalize("NFKD", str(nome))  
5      t = "".join(c for c in t if not unicodedata.combining(c))  
6      t = t.replace("ç", "c")  
7      return t.strip().lower()  
8  
9  def _slug(s: str) -> str:  
10     s = unicodedata.normalize("NFKD", str(s))  
11     s = "".join(c for c in s if not unicodedata.combining(c))  
12     s = re.sub(r"[^a-zA-Z0-9]+", "_", s.strip().lower())  
13     return s.strip("_")
```



COMO OS CSVS FORAM CONSTRUIDOS

- **bairros_unique.csv**
 - Feito para facilitar a análise, partir de **bairros_recife.csv**
- **adjacencias_bairros.csv**
 - Feito para representar adjacências entre os bairros de Recife, contendo os pesos associados.
 - Com base nos mapas da Prefeitura do Recife e Google Maps para determinar rotas e pesos (distância em km)
- **enderecos.csv**
 - Foi escolhido 6 diferentes pares de (rua, bairro).



ALGORITMOS



```
1 def calcular_distancias():
```

Monta o grafo a partir de
adjacencias_bairros.csv

Lê **enderecos.csv**

Calcula o custo e o melhor caminho para cada par de bairro

Cria um json com o percurso de cada par de bairro

Cria um csv com o custo e caminho para cada par de bairro



```
1 def dijkstra_path(  
2     grafo: Graph,  
3     origem: str,  
4     destino: str  
5 ) -> List[str]:
```

Calcula e retorna o melhor caminho para um par (origem, destino)



```
1 def dijkstra_path_length(  
2     grafo: Graph,  
3     origem: str,  
4     destino: str  
5 ) -> float:
```

Calcula e retorna o menor custo para um par (origem, destino)

ESCALABILIDADE

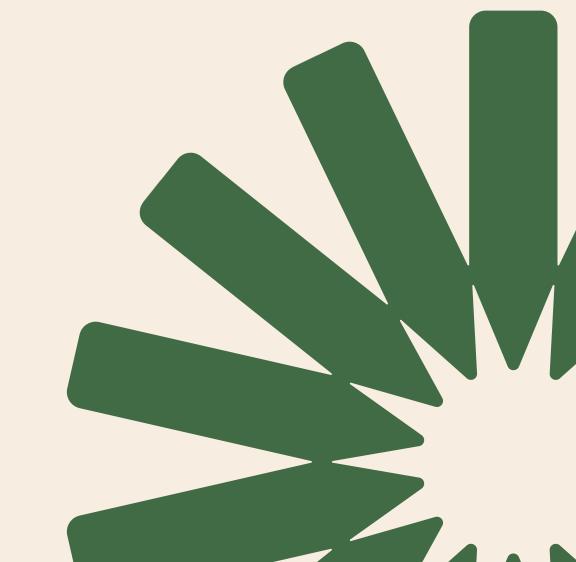
$$O((V + E) * \log(V))$$

- **V** é o número de vértices (bairros).
- **E** é o número de arestas (conexões/ruas).
- **Dijkstra** não percorre todos os caminhos cegamente.
- **Sempre** escolhe o nó mais promissor (menor custo acumulado) da fila

O tempo cresce um pouco mais que o dobro, se a quantidade de vértices (bairros), por exemplo, dobrar.

DATASET PARTE 2



- Ligações Rodoviárias e Hidroviárias - IBGE 2016
 - Estrutura de **Grafo Direcionado**
 - **Municípios** como Vértices (nomemun_a, nomemun_b)
 - **Tempo** como Peso nas Arestas (em segundos)
 - *Para cumprir os requisitos do algoritmo de Bellman-Ford, foram atribuídos pesos negativos a determinados vértices:
 - Palmitos → São Carlos → São José do Rio Preto → Palmitos
 - *No contexto do dataset não faz sentido ter pesos negativos
- 



COMO OS CSVS FORAM CONSTRUÍDOS PT2

- **enderecos_parte2.csv**

- Foi escolhido 5 diferentes pares de (municipio_inicio, municipio_destino) para utilizar no algoritmo dijkstra, bfs e dfs.

- **enderecos_bellmanford.csv**

- Foi escolhido 2 diferentes pares de (municipio_inicio, municipio_destino) para utilizar no algoritmo bellman-ford.



ALGORITMO DE DIJKSTRA

- O código do algoritmo Dijkstra da Parte 1 foi reaproveitado na Parte 2
- A única modificação técnica foi realizada na inicialização do objeto grafo, de **não direcionado** para **direcionado**
- Lê LRH2016_00_Base_Completa.csv para montar o grafo
- Lê enderecos_parte2.csv para obter a lista de rotas

```
G = Graph(direcionado=True)
```



ALGORITMO DE BELLMAN-FORD

```
def bellman_ford(  
    G: Graph,  
    source: str,  
    target: str,  
    weight: str | Callable = "weight",  
):
```

```
def bellman_ford_path(  
    G: Graph,  
    source: str,  
    target: str,  
    weight: str | Callable = "weight",  
) -> List[str]:
```

Encontrar o caminho mais curto de uma origem a um destino

Repete o processo de melhorar (relaxar) as distâncias dos nós vizinhos

Retorna os dicionários de distâncias (dist) e predecessores (pred) com todos os nós

Se o destino for alcançável, reconstrói o caminho usando o dicionário de predecessores

Retorna uma lista de nós que compõem o caminho mais curto

```
def bellman_ford_path_length(  
    ...)
```

Calcula o custo do caminho mais curto



ALGORITMO DE BELLMAN-FORD



```
1 def calcular_distancias_bellman_ford():
```



```
1 def verificar_ciclo_no_caminho(caminho):
```

— Lê LRH2016_00_Base_Completa.csv para montar o grafo

— Lê enderecos_bellmanford.csv para obter a lista de rotas

— Calcula o custo e o melhor caminho (e detecta ciclo negativo) para cada par de municípios

— Cria um json com o percurso de cada par de municipios

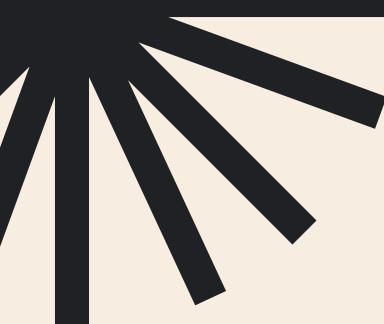
— Cria um csv com o custo e caminho para cada par de municipios

— Analisa a estrutura de uma lista de nós para verificar se existe ou não um **ciclo completo**



```
1 def analisar_ciclo_negativo(  
2     G,  
3     caminho,  
4     custo,  
5     origem,  
6     destino,  
7     excecao_ciclo=False  
8 ):
```

— Determina se uma rota calculada pelo Bellman-Ford configura um **ciclo negativo**



ALGORITMO DE BFS

```
def bfs_ordem_camadas_ciclos_dir(  
    G: Graph,  
    source: str,  
    max_cycles: int = 10,  
):
```

Faz busca em largura em um grafo direcionado a partir do nó source.

Visita os nós por camadas de distância (arestas) usando uma fila.

Para cada nó visitado, armazena o pai e a distancia a partir da origem

Ao encontrar arestas que levam a nós já visitados, reconstroí ciclos usando o caminho até o ancestral comum

Ao final, será feito uma reorganização dos nós em camadas de profundidade

— Retorna uma tupla para cada par definido em enderecos_parte2.csv:

- a ordem: sequência de visita dos nós
- as camadas: nós agrupados por distância (conta os passos do bfs)
- os ciclos: se forem detectados irá retornar os ciclos direcionados utilizados.



ALGORITMO DE DFS

```
def dfs_ordem_camadas_ciclos_dir(  
    G: Graph,  
    source: str,  
    max_cycles: int = 10,  
):
```

- Faz busca em profundidade em um grafo direcionado a partir do nó source, usando uma pilha
- Para cada nó visitado, guarda o pai e a profundidade em relação à origem
- Ao encontrar arestas que levam a nós já visitados, reconstroí ciclos usando o caminho até o ancestral comum.
- Ao terminar a busca, chama _montar_camadas_dfs para organizar os nós por profundidade em camadas.

— Retorna uma tupla para cada par definido em enderecos_parte2.csv:

- ordem: sequência da descoberta dos nós
- camadas: níveis de profundidade
- ciclos: ciclos direcionados detectados

MÉTRICAS

Cálculo:

Pontos de partida: Santo Antônio do Sudoeste, Planura e Wanderlândia.

Dentre as métricas: tempo necessário para executar o caminho e memória consumida

Conclusões:

O Dijkstra apresentou desempenho bastante variado conforme o par de origem e destino, não comporta pesos negativos e consumiu mais memória.

O Bellman-Ford apresentou tempos maiores, porém uso de memória mais constante e lidou muito bem com nossos pesos, inclusive negativos.



Melhor para nosso contexto!

O DFS apresentou tempos extremamente reduzidos, porém visitou apenas frações pequenas do grafo, por seguir um aprofundamento único antes de ramificar.

O BFS percorreu praticamente todo o grafo nas três execuções, com tempo muito baixo e uso de memória bastante estável, porém não lidou tão bem com os pesos.

FRONT INTERATIVO





OBRIGADO A TODOS!