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)

```python
def _normalize(nome: str) -> str:
    if nome is None:
        return ""
    t = unicodedata.normalize("NFKD", str(nome))
    t = "".join(c for c in t if not unicodedata.combining(c))
    t = t.replace("ç", "c")
    return t.strip().lower()

def _slug(s: str) -> str:
    s = unicodedata.normalize("NFKD", str(s))
    s = "".join(c for c in s if not unicodedata.combining(c))
    s = re.sub(r"[^a-zA-Z0-9]+", "_", s.strip().lower())
    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

```python
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

```python
def dijkstra_path(
    grafo: Graph,
    origem: str,
    destino: str
) -> List[str]:
```

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

```python
def dijkstra_path_length(
    grafo: Graph,
    origem: str,
    destino: str
) -> 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 CONSTRUIDOS PT2

- **enderecos_parte2.csv**
  - Foi escolhido 5 diferentes pares de (municipio_inicio, municipio_destino) para utilizar no algorítimo dijikstra, bfs e dfs.
- **enderecos_bellmanford.csv**
  - Foi escolhido 2 diferentes pares de (municipio_inicio, municipio_destino) para utilizar no algorítimo 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

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

- 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

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

- Se o destino for alcançável, reconstrói o caminho usando o dicionário de predecessores
- Retorna uma lista de nós que compõem o caminho mais curto

```python
def bellman_ford_path_length(
```

- Calcula o custo do caminho mais curto

ALGORITMO DE BELLMAN-FORD

```python
def calcular_distancias_bellman_ford():
```

- 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

```python
def verificar_ciclo_no_caminho(caminho):
```

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

```python
def analisar_ciclo_negativo(
    G,
    caminho,
    custo,
    origem,
    destino,
    excecao_ciclo=False
):
```

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

ALGORITMO DE BFS

```python
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 distãncia (conta os passos do bfs)
- os ciclos: se forem detectados irá retornar os ciclos direcionados utilizados.

ALGORITMO DE DFS

```python
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!