UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
CENTRO TECNOLÓGICO – CTC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA – INE

RELATÓRIO T1

DISCIPLINA: GRAFOS - INE5413
PROFESSOR: RAFAEL DE SANTIAGO

ALUNOS: ARTUR BARICHELLO (16200636)
LETICIA DO NASCIMENTO (16104595)
MARCOS AURÉLIO LESSA (17100526)

FLORIANÓPOLIS, 19 DE SETEMBRO DE 2019

1. Introdução

Para a realização do trabalho, escolhemos Python como linguagem de programação para o desenvolvimento, por questão de afinidade dos membros da equipe com a mesma.

Optamos por criar o Grafo como sendo uma *namedtuple*, uma espécie de estrutura em Python. Como se pode observar na imagem abaixo, esta estrutura nomeada de *Graph*, tem como atributos: *num_vertices, conj_vertices, conj_arestas,* que, respectivamente, representam o numero de vértices presente no Grafo, o conjunto de vértices, e o conjunto de arestas do mesmo.

```
# Cria o grafo propriamente dito
Graph = namedtuple("Graph", "num_vertices conj_vertices conj_arestas")
```

2. Vértices

Um vértice foi definido como uma tupla, onde o primeiro elemento indica o índice do vértice e o segundo elemento o seu rótulo. Na imagem abaixo, o vértice indica que seu índice é 1 e seu rótulo ‘Itajai’.

```
(1, 'Itajai')
```

Já o conjunto de vértices, é uma lista que tem como tamanho igual ao atributo *num_vertices.* Esta lista contém todos os vértices do Grafo lido, como por exemplo:

```
[(1, 'Itajai'), (2, 'Camboriu'), (3, 'Brusque'), (4, 'Nova Trento'), (5, 'São João Batista'), (6, 'Navegantes')]
```

3. Arestas

Decidimos criar uma lista bidimensional para o conjunto das arestas, onde cada lista interna contém os vértices vizinhos de um determinado vértice. Ou seja, dado o exemplo dos vértices acima, a primeira posição da lista externa conterá todos os vértices com os quais o vértice (1, ‘Itajai”) se conecta, formando uma aresta. Optamos fazer assim, pois, quando for necessário receber os vértices vizinhos de algum outro, acessamos a posição inicial da sua lista de vizinhos em O(1), e resgatamos todos os vértices vizinhos a ele em O(n), sendo n o número de vértices os quais se conectam com ele. Essa solução é vista com muito mais eficiência do que se tivéssemos criado uma lista com todas as arestas existentes no Grafo, pois para um Grafo com muitas arestas, levaria muito mais tempo para conseguir acessá-las.

```
[[((2, 'Camboriu'), 1.0), ((6, 'Navegantes'), 1.0)],
```

A imagem acima mostra a primeira posição da lista externa, representada pelo vértice (1, ‘Itajai’). A lista interna nessa posição, indica os vértices com os quais o vértice (1, ‘Itajai’) se conecta, formando uma aresta. Cada elemento que se conecta com o vértice mencionado é representado por uma tupla, onde na primeira posição se encontra o vértice e na segunda o peso da aresta que conecta os dois vértices.

4. Busca em largura

Utilizamos 3 dicionários (estrutura de dados equivalente ao Hash Map em outras linguagens de programação). Um para vértices visitados, onde a chave é um vértice e o valor é boolean. Um segundo dicionário para a distância do vértice inicial, onde o valor é um número float. E o terceiro sendo uma árvore de ancestrais.

E além disso, foi usado uma fila para representar o vértice que chamamos de *u*, no loop do algoritmo.

```
def bfs(n_vertices: int) -> Tuple[dict, dict]:
    visited: dict = {}
    distance: dict = {}
    predecessor: dict = {}
    for v in Graph.conj_vertices:
        i = v[0]
        visited[i] = False
        distance[i] = float("inf")
        predecessor[i] = None

    start = Graph.conj_vertices[0]
    visited[start[0]] = True
    distance[start[0]] = 0

    queue: list = [start]
```

5. Ciclo euleriano

Usamos *set* (conjunto), tupla e dicionário (ou Hash map). A tupla representa uma aresta, onde cada elemento desta é um vértice. O conjunto, contém todas as arestas. E o dicionário para atribuir quais arestas foram ou não visitadas. Se uma aresta tiver o valor *False*, quer dizer que ainda não foi visitada, caso tiver *True,* já foi.

```
ord = set(map(tuple, map(sorted, arestas)))
visitado = {aresta: False for aresta in ord}
```

6. Floyd-Warshall

Neste algoritmo, utilizamos uma matriz bidimensional para representar as distâncias através de um número float.

```
dist: List[List[float]] = [
    [inf for i in range(n_vertices)] for j in range(n_vertices)
]
```

7. Dijkstra

Aqui, foi usado a estrutura *set,* onde contém o conjunto de vértices. Também utilizamos dois dicionários: o primeiro que chamamos de *distances,* para as distâncias, e o segundo chamado de *paths*, onde cada vértice está relacionado com um array de vértices.

```
def dijkstra(initial: int):
    vertices = set(Graph.conj_vertices)

    # initiate distance, path -> inf, none for all vertices
    distances: dict = {vertex: inf for vertex in vertices}
    paths: dict = {vertice: [] for vertice in vertices}

    # prepare initial vertice attributes
    initial_vertex = getVertexByNumber(initial)
    distances[initial_vertex] = 0
    paths[initial_vertex] = [initial]
```