

Explorando a linguagem Python e a biblioteca NetworkX para a solução de problemas em grafos

UENO, Caio Luiggy Ryoichi Sawada

Graduando em Ciência da Computação – UFSCar

LEITE, João Augusto

Graduando em Ciência da Computação – UFSCar

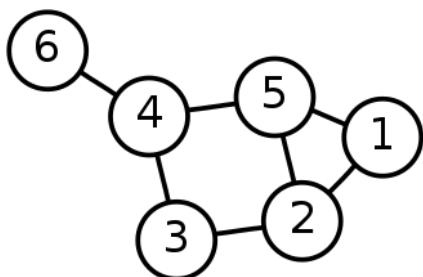
MOURA, Daniel Lúcio

Graduando em Ciência da Computação – UFSCar

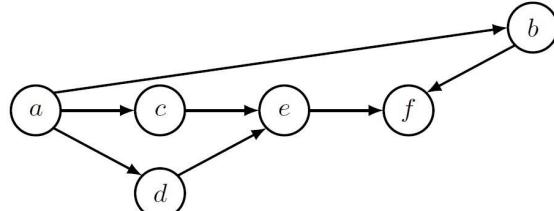
INTRODUÇÃO

A teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. O artigo de *Leonhard Euler*, publicado em 1736, sobre o problema das sete pontes de Königsberg, é considerado o primeiro resultado da teoria dos grafos. Além disso, o primeiro livro didático sobre teoria dos grafos foi escrito por Dénes König e publicado em 1936. Apesar de parecer antiga, a teoria dos grafos é relativamente nova, já que a matemática é uma área já estudada há milênios. Aliada à computação, a teoria dos grafos tornou-se uma ferramenta importantíssima para modelar problemas reais.

A estrutura denominada “grafo” pode ser representada por uma tupla $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. Um vértice é uma estrutura que pode ser ligada a outros vértices por meio de uma aresta. Dependendo da aplicação, arestas podem ou não ter direção. Se as arestas têm uma direção associada, então o grafo em questão é chamado “Dígrafo”.



Grafo

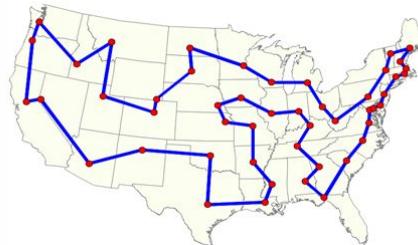


Dígrafo

A teoria dos grafos é extremamente útil para modelar certos problemas do mundo real, como por exemplo: redes sociais, redes de computadores, problemas de pareamento, análise de rotas de trânsito, ranqueamento de páginas na internet e etc.



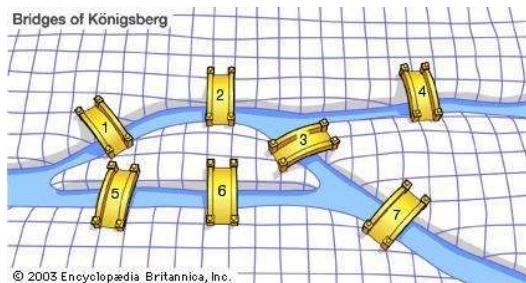
Redes Sociais



Caminhos mínimos

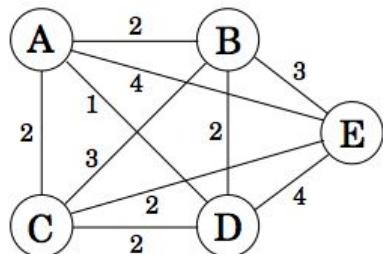
Alguns problemas clássicos da teoria dos grafos:

- Sete pontes de Königsberg



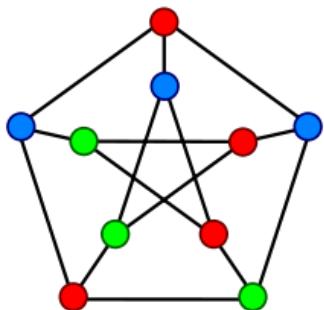
Objetivo: atravessar todas as pontes sem passar por uma mesma ponte mais de uma vez.

- Problema do caixeiro-viajante



Objetivo: encontrar a menor rota que passa por todos os vértices sem passar por um mesmo vértice mais de uma vez.

- Coloração de vértices



Objetivo: encontrar o menor número de cores necessárias para separar os vértices em grupos cujos vértices de um mesmo grupo não estão conectados.

Tendo em vista esse ambiente de propriedades interessantes e a importância que os grafos têm para a computação nos dias de hoje é que se decidiu explorar alguns e implementar possíveis soluções para os mesmos.

A linguagem usada para a confecção deste artigo foi o Python¹, uma linguagem muito robusta em relação ao manuseio de grafos, devido principalmente a sua biblioteca NetworkX², que foi amplamente utilizada na implementação de todos os problemas propostos. Devido a sua característica de ser interpretada, a linguagem Python fornece uma simplicidade durante as execuções dos algoritmos, além de ser de fácil entendimento mesmo para quem não possui conhecimentos avançados em programação. É de fato uma ótima ferramenta para trabalhos mais robustos, mas é encorajado, caso haja interesse, reproduzir os resultados em uma linguagem diferente para comparação.

Quanto à biblioteca NetworkX, desde a criação dos grafos até a plotagem de seus vértices e arestas foram feitas através dos métodos presentes nesta, e muitos resultados obtidos podem ser conferidos em métodos que a própria biblioteca fornece - como o algoritmo de Prim, BFS, DFS e Dijkstra.

O artigo apresenta a fundamentação teórica dos problemas escolhidos para serem resolvidos, bem como a implementação das resoluções em Python, seguido dos resultados de cada problema - plotagem de grafos, matrizes e tabelas. Por fim uma seção sobre os materiais e os ambientes usados durante a execução do artigo e outra contendo uma visão final sobre os resultados e o objetivo deste artigo.

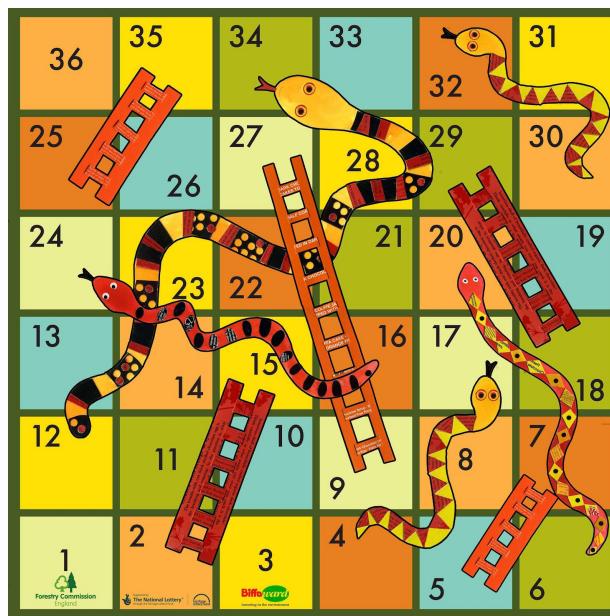
FUNDAMENTAÇÃO TEÓRICA

Nesta seção será abordado quais os problemas relevantes selecionados na teoria dos grafos e como os problemas foram atacados, explicando as estratégias e possível solução dos mesmos, e também o código implementado.

Problema 1:

Snakes and Ladders (Cadeias de Markov e Random-Walks)

Snakes and Ladders é um famoso jogo de tabuleiro em que a cada rodada um jogador joga uma moeda não viciada e avança 1 casa se obtiver cara ou avança 2 casas se obtiver coroa. Se o jogador para no pé da escada, então ele imediatamente sobe para o topo da escada. Se o jogador cai na boca de uma cobra então ele imediatamente escorrega para o rabo. O jogador sempre inicia no quadrado de número 1. O jogo termina quando ele atinge o quadrado de número 36.



Esse jogo pode ser modelado como uma Cadeia de Markov, em que cada casa é um vértice e cada possibilidade de movimento é uma aresta. Todos os vértices se ligam ao vértice seguinte e ao próximo (cara e coroa na moeda, respectivamente), porém, os vértices que possuem escadas ou cobras precisam ligar-se conforme a imagem. Além disso, o vértice final (36) deve ser um loop, indicando que o jogador chegou à posição final.

Com base nas informações, são dadas 3 questões:

- Especificar a matriz de transição de estados P que define a função de transição da cadeia de Markov Homogênea.
- Desenvolver um script em Python para calcular a distribuição estacionária da cadeia de Markov homogênea em questão. Qual é a probabilidade de um jogador vencer o jogo, ou seja, qual a probabilidade de se atingir o estado 36 no longo

prazo? Considere $k = 100$ um número suficiente de iterações no Power Method. Qual o estado mais provável de ser acessado?

c) Especificar a matriz \bar{P} referente ao modelo Pagerank considerando $\alpha = 0.1$. Considerando $k = 100$, aplique o Power method e compare o resultado com o obtido no item b). As distribuições estacionárias obtidas em b) e c) são iguais ou diferentes?

Primeiramente, inicializa-se o dígrafo usando uma função que gera as arestas da forma $E_i = \{(V_i, V_{i+1}), (V_i, V_{i+2})\}$, ou seja, liga todo vértice aos 2 próximos vértices. Há uma exceção para o vértice 36, que é o último, e para o vértice 35, que só se liga ao 36.

```
In [2]: 1 def gen_edges():
2     """Gera a lista de arestas para o digrafo"""
3     edges = []
4     for i in range(1,36):
5         edges.append((i,i+1))
6         if (i+2 <= 36):
7             edges.append((i, i+2))
8     return edges
```

Depois, adiciona-se todas as arestas que representam as escadas e as cobras e remove-se as arestas que impediriam o movimento obrigatório para escada ou cobra. Fazendo com que o vértice que possua escada/cobra tenha somente uma ligação: o vértice final da escada/cobra. Por fim, adiciona-se o loop no vértice 36.

```
In [3]: 1 G = nx.DiGraph()
2 G.add_nodes_from(range(1,37)) # Adiciona 36 vértices no grafo G
3 G.add_edges_from(gen_edges()) # Adiciona as arestas entre os vértices
4 G.add_edges_from([(2,15), (5,7), (9,27), (18,29), (25,35)]) # Adiciona as arestas 'Ladders'
5 G.add_edges_from([(17,4), (20,6), (32,30), (34,12), (24,16)]) # Adiciona as arestas 'Snakes'
6 G.add_edge(36,36)
7 G.remove_edges_from([(2,3),(2,4),(5,6),(9,10),(9,11),
8                 (17,18),(17,19),(18,19),(18,20),
9                 (20,21),(20,22),(24,25),(24,26),
10                (25,26),(25,27),(32,33),(32,34),
11                (34,35),(34,36)])
12 G.out_degree()
```

Out[3]: OutDegreeView({1: 2, 2: 1, 3: 2, 4: 2, 5: 1, 6: 2, 7: 2, 8: 2, 9: 1, 10: 2, 11: 2, 12: 2, 13: 2, 14: 2, 15: 2, 16: 2, 17: 1, 18: 1, 19: 2, 20: 1, 21: 2, 22: 2, 23: 2, 24: 1, 25: 1, 26: 2, 27: 2, 28: 2, 29: 2, 30: 2, 31: 2, 32: 1, 33: 2, 34: 1, 35: 1, 36: 1})

Obs: OutDegreeView é o grau de saída para cada vértice.

O próximo passo consiste em gerar a matriz P , que define as transições de estados.

$$P = \Delta^{-1} A \quad \text{onde} \quad \Delta^{-1} = \begin{bmatrix} \frac{1}{d(v_1)} & 0 & 0 & 0 \\ 0 & \frac{1}{d(v_2)} & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \frac{1}{d(v_n)} \end{bmatrix}$$

Gerando a matriz A (Matriz de adjacências):

```
In [4]: 1 adj_matrix = nx.adjacency_matrix(G).toarray() # Gera a matriz de adjacencias de G
2 adj_matrix
Out[4]: array([[0, 1, 1, ..., 0, 0, 0],
   [0, 0, 0, ..., 0, 0, 0],
   [0, 0, 0, ..., 0, 0, 0],
   ...,
   [0, 0, 0, ..., 0, 0, 0],
   [0, 0, 0, ..., 0, 0, 1],
   [0, 0, 0, ..., 0, 0, 1]], dtype=int64)
```

Gerando a matriz Δ^{-1} :

```
In [5]: 1 delta = np.zeros((36,36))
2 np.fill_diagonal(delta,gen_inversedegree(G)) #Gera a matriz delta
3 delta.diagonal() #Diagonal de Delta
Out[5]: array([0.5, 1., 0.5, 0.5, 1., 0.5, 0.5, 0.5, 1., 0.5, 0.5, 0.5, 0.5,
   0.5, 0.5, 0.5, 1., 1., 0.5, 1., 0.5, 0.5, 0.5, 0.5, 1., 1., 0.5,
   0.5, 0.5, 0.5, 0.5, 1., 0.5, 1., 1., 1.])
```

Obs: Foi definida uma função `gen_inversedegree` que retorna uma lista com o inverso do grau de cada vértice.

```
def gen_inversedegree(G):
    """Gera uma lista com o inverso dos graus de um grafo"""
    degrees_G = []
    for d in dict(G.out_degree()).values():
        degrees_G.append(1/d)

    return degrees_G
```

Com isso, é possível gerar a matriz P através de uma simples multiplicação de matrizes.

Agora, para cada $P(i,j)$ da matriz P, temos a probabilidade do vértice i transicionar para o vértice j .

Com isso, será possível calcular a distribuição estacionária dessa cadeia de markov. A distribuição estacionária revela a probabilidade de se atingir um estado qualquer passado um período grande de "tempo".

Primeiramente define-se o vértice inicial w_0 como o vértice 0 (casa 1 do tabuleiro). Basta atribuir 100% de chances de ir para o vértice 0 e 0% para os demais.

Depois, fazemos o Power Method, que consiste em elevar a matriz P a uma potência grande (k) e depois multiplicar w_0 por P^k .

```
def power_method(P,w,it):
    """Calcula a distribuicao estacionaria usando o power method"""
    return (w @ np.linalg.matrix_power(P,it))
```

A parte final deste capítulo, consiste em gerar a matriz \bar{P} , ou Google Matrix, que corresponde ao modelo Pagerank. O modelo Pagerank é similar ao Random Walk acima, porém, são tratados os casos em que é impossível acessar ou sair de determinado vértice. Para isso, adiciona-se a todos os vértices uma pequena chance de salto para um vértice qualquer.

A matriz \bar{P} pode ser calculada da seguinte maneira:

$\bar{P} = (1-\alpha)P + \alpha \frac{1}{n}U$ (Google matrix) onde

$$U = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \dots & \dots & \dots & \dots \\ 1 & 1 & \dots & 1 \end{bmatrix}$$

denota uma matriz $n \times n$ de 1's e $\alpha \in [0,1]$

Em Python:

```
def Google_matrix(P, alfa):
    """Gera a matriz P barra (google matrix)"""
    P_Barra = (1-alfa)*P + (alfa/P.shape[0])*np.ones((36,36))
    return P_Barra
```

Problema 2:

Árvore Geradora Mínima

Dado um Grafo $G = (V, E)$ e um vértice inicial $v \in V$, obter a árvore geradora mínima para G a partir de v , ou seja, obter um grafo $T = (V, A)$, onde $A \subseteq E$ tal que seja uma árvore - grafo conexo e acíclico - que minimiza $\sum \omega(e)$, $e \in A$. Tendo em vista esse objetivo, foi implementado um algoritmo baseado no algoritmo de Prim para *Minimum Spanning Tree*, que possui como entrada um grafo $G = (V, E)$ e um vértice $v \in V$. A seguir a imagem da função `MST_Prim` definida:

```

def MST_Prim(G,v_inicial):
    nx.set_node_attributes(G, np.Inf, 'lambda') # setar lambda
    nx.set_node_attributes(G, None, 'pi') # setar pi

    lamb = {i: np.Inf for i in G.nodes().keys()} #dicionario do lambda dos vertices
    lamb[v_inicial] = 0 #lambda do v_inicial recebe 0
    G.node[v_inicial]['lambda'] = 0

    while bool(lamb): #enquanto houver vertice no dicionario
        u = min(lamb, key=lamb.get)

        for v in nx.neighbors(G,u): #vizinha de u
            if v in lamb.keys() and G.edges[v,u]['weight'] < lamb[v]: #se aresta(u,v) < lambda[v]
                #e v ainda estiver no dic
                lamb[v] = G.edges[v,u]['weight'] #atualiza o lambda
                G.node[v]['lambda'] = G.edges[v,u]['weight']
                G.node[v]['pi'] = u #atualiza o pi - pai do vertice

        del lamb[u] #tira o vertice u do dicionario

    resultados = {} #dicionario com os resultados
    resultados['pi'] = nx.get_node_attributes(G,'pi')
    resultados['lambda'] = nx.get_node_attributes(G,'lambda')

    Pr = pd.DataFrame(resultados) #Dataframe dos resultados
    MST = nx.Graph() #Criação da arvore - MST
    MST.add_nodes_from(G) #adição dos vertices

    for k in G.nodes().keys(): #Criacao das arestas na arvore a partir do pi
        if k != v_inicial:
            MST.add_edge(k,resultados['pi'][k], weight=resultados['lambda'][k])

    return (Pr,MST) #retorna uma tupla com o dataframe dos resultados e a arvore

```

Primeiro são definidos dois atributos para os vértices de G , λ e π respectivamente. O λ corresponde ao custo para “sair” de cada vértice e construir a árvore. O π representa o “pai” do vértice, se $\pi(u) = v$, então o vértice v é pai do vértice u , ou seja, existe aresta (u, v) na árvore. Para facilitar a manipulação durante a execução do algoritmo foi criado um dicionário - lamb - que também armazena o valor do λ de cada vértice, entretanto, ao fim da execução o dicionário estará vazio. Todos os vértices ainda não possuem pai e seus valores para λ são inicializados com infinito, exceto o vértice inicial, como mostra a figura a seguir:

```

nx.set_node_attributes(G, np.Inf, 'lambda') # setar lambda
nx.set_node_attributes(G, None, 'pi') # setar pi

lamb = {i: np.Inf for i in G.nodes().keys()} #dicionario do lambda dos vertices
lamb[v_inicial] = 0 #lambda do v_inicial recebe 0
G.node[v_inicial]['lambda'] = 0

```

Após a inicialização das variáveis, é construído o laço principal do algoritmo: enquanto houver vértices no dicionário lamb , escolha o vértice com o menor valor de λ , e para cada vizinho v do vértice u escolhido atualizar o valor de $\lambda(v)$ e $\pi(v)$ se

o custo da aresta (u, v) for menor que o $\lambda(v)$. Após todos os vizinhos serem consultados, deletar o vértice u do dicionário:

```
while bool(lamb): #enquanto houver vertice no dicionario
    u = min(lamb, key=lamb.get)

    for v in nx.neighbors(G,u): #vizinhança de u
        if v in lamb.keys() and G.edges[v,u]['weight'] < lamb[v]: #se aresta(u,v) < lambda[v]
            lamb[v] = G.edges[v,u]['weight'] #atualiza o lambda
            G.node[v]['lambda'] = G.edges[v,u]['weight']
            G.node[v]['pi'] = u #atualiza o pi - pai do vertice

    del lamb[u] #tira o vertice u do dicionario
```

Por fim é criado um dicionário com os λ e π de todos os vértices que depois é transformado em um Dataframe - tabela. É criado a árvore - MST - e adicionada a ela os vértices de G . Há um laço para cada vértice de G que adiciona uma aresta entre um vértice e seu pai - $\pi(v)$. Retorna uma tupla contendo o Dataframe e a árvore MST :

```
resultados = {} #dicionario com os resultados
resultados['pi'] = nx.get_node_attributes(G,'pi')
resultados['lambda'] = nx.get_node_attributes(G,'lambda')

Pr = pd.DataFrame(resultados) #Dataframe dos resultados
MST = nx.Graph() #Criação da arvore - MST
MST.add_nodes_from(G) #adição dos vertices

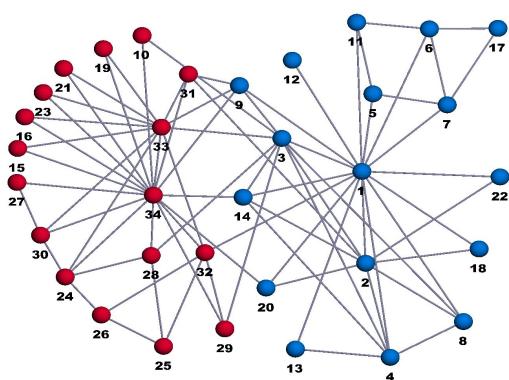
for k in G.nodes().keys(): #Criacao das arestas na arvore a partir do pi
    if k != v_inicial:
        MST.add_edge(k,resultados['pi'][k], weight=resultados['lambda'][k])

return (Pr,MST) #retorna uma tupla com o dataframe dos resultados e a arvore
```

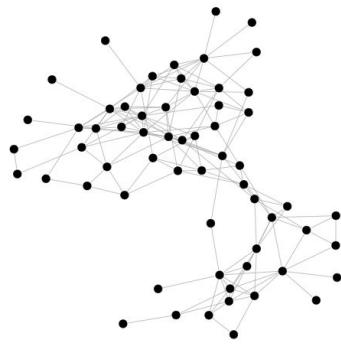
Problema 3:

Busca em Grafos

O terceiro problema envolve a implementação de um algoritmo de busca em largura e outro de busca em profundidade em dois grafos diferentes, Zarachy e Dolphins.



Zachary's Karate Club

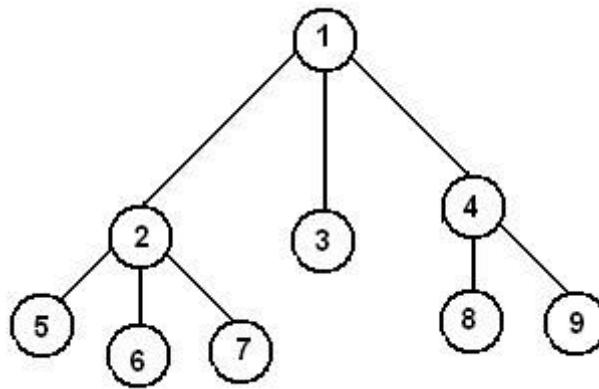


Dolphins Social Network

Primeiramente, há de se explicar o que é uma busca em um grafo e porque um tipo de busca é diferente do outro.

O objetivo de uma busca genérica é encontrar todos os vértices de um grafo. Obviamente, isso depende de como é o grafo. Se ele não for conexo, por exemplo, não será possível visitar todos os elementos do grafo.

Uma busca em um grafo consiste basicamente em extrair uma árvore do mesmo, já que em uma árvore, existe um único caminho entre 2 vértices quaisquer. Apesar de no grafo original, possivelmente existir mais de 1 caminho entre 2 vértices, só é necessário que haja 1 para que possamos visitá-lo.



Ou seja, enquanto navega-se no grafo, cada nó visitado é "adicionado" à árvore, que aos poucos conterá todos os vértices do grafo. Esse princípio ocorre tanto na busca em largura quanto na busca em profundidade. A diferença entre os dois métodos consiste na maneira em que os vértices são visitados.

Busca em Largura

Na busca em largura, utiliza-se uma estrutura de dados de fila, em que o primeiro elemento a entrar será o último a sair. Além disso, são utilizados "atributos"

que ajudarão a categorizar os vértices conforme caminhamos no grafo. Os atributos são:

- Cor:
 - Branco: vértice ainda não visitado;
 - Cinza: vértice já visitado;
 - Preto: vértice já saiu da fila.
- λ : menor distância entre o vértice e a raiz.
- π : antecessor do vértice.

A inicialização do algoritmo consiste em marcar todos os vértices como branco, seus π como nulo, seus λ como ∞ e inserir um vértice inicial v_0 na fila (que possui $\lambda = 0$).

```

nx.set_node_attributes(G,np.inf,'lambda') # Menor distancia do vertice v ate a raiz
nx.set_node_attributes(G,'white','color') # white == v ainda nao visitado
                                              # gray == v ja visitado
                                              # black == todos os vizinhos de v ja foram visitados
nx.set_node_attributes(G,None,'pi') # predecessor de v

G.node[str(v_inicial)]['color'] = 'gray'
G.node[str(v_inicial)]['lambda'] = 0
G.node[str(v_inicial)]['pi'] = None
fila = []

fila.append(str(v_inicial))

```

Depois, quando o algoritmo começa de fato, retira-se o único elemento da lista (v_0) insere-se seus vizinhos na fila (se eles ainda não foram visitados). Quando o laço se inicia novamente, o elemento que sairá da fila será um dos vizinhos de v_0 , que será marcado como cinza, terá seu λ atualizado para $\lambda(v_0) + 1$ e seu π será v_0 . Quando todos os vizinhos de v_0 estiverem na fila e já tiverem sido visitados (cinzas), v_0 será marcado como preto. Este processo se repete até que a fila fique vazia.

```

while(len(fila) != 0):
    u = fila.pop()
    for v in nx.neighbors(G,str(u)): # Itera sobre os vizinhos de u
        if G.node[str(v)][‘color’] == ‘white’:
            G.node[str(v)][‘lambda’] = G.node[str(u)][‘lambda’] + 1
            G.node[str(v)][‘pi’] = str(u)
            G.node[str(v)][‘color’] = ‘gray’
            fila.append(str(v))
    G.node[str(u)][‘color’] = ‘black’

```

Pela característica própria da fila, garante-se que todos os vértices de um nível i serão visitados antes dos vértices de nível $j > i$.

Ao final do algoritmo, teremos o λ de cada vértice, que representa seu nível na árvore, ou distância dele até v_0 , o mapa de predecessores (π), que estrutura a

árvore de fato, mostrando qual vértice antecede outro e, finalmente, todos os vértices terão a cor preta.

```
def BFS(G,v_inicial):
    """Faz uma busca em largura em um grafo"""
    nx.set_node_attributes(G,np.inf,'lambda') # Menor distancia do vertice v ate a raiz
    nx.set_node_attributes(G,'white','color') # white == v ainda nao visitado
                                                # gray == v ja visitado
                                                # black == todos os vizinhos de v ja foram visitados
    nx.set_node_attributes(G,None,'pi') # predecessor de v

    G.node[str(v_inicial)]['color'] = 'gray'
    G.node[str(v_inicial)]['lambda'] = 0
    G.node[str(v_inicial)]['pi'] = None
    fila = []

    fila.append(str(v_inicial))

    while(len(fila) != 0):
        u = fila.pop()
        for v in nx.neighbors(G,str(u)): # Itera sobre os vizinhos de u
            if G.node[str(v)]['color'] == 'white':
                G.node[str(v)]['lambda'] = G.node[str(u)]['lambda'] + 1
                G.node[str(v)]['pi'] = str(u)
                G.node[str(v)]['color'] = 'gray'
                fila.append(str(v))
        G.node[str(u)]['color'] = 'black'

    # Retorna os resultados da busca
    resultados = {}
    resultados['pi'] = nx.get_node_attributes(G,'pi')
    resultados['lambda'] = nx.get_node_attributes(G,'lambda')
    resultados['color'] = nx.get_node_attributes(G,'color')

    T = nx.Graph() # arvore a ser retornada
    for node in G.nodes().keys(): # Criacao das arestas na arvore
        T.add_edge(node,resultados['pi'][node])
    return (pd.DataFrame(resultados),T)
```

Visão geral do algoritmo

Busca em Profundidade

A grande diferença entre a busca em largura e a busca em profundidade é a estrutura de dados, que no caso da busca em profundidade é uma pilha, ou seja, o primeiro elemento a entrar é o primeiro elemento a sair. Computacionalmente falando, nem é necessário implementar a pilha, já que é possível utilizar a pilha de recursão para isso.

Há também uma pequena mudança nos atributos dos vértices, que não possuem mais λ . Agora, todo vértice terá, além de π e da coloração, os atributos d e f , que correspondem ao instante de entrada e ao instante de saída do vértice, respectivamente. O tempo será uma medida importante para o algoritmo, portanto, utiliza-se uma variável global que será incrementada durante o processo.

A inicialização do algoritmo também é semelhante à busca em largura, porém, nos novos atributos, deve-se inicializar o tempo, d e f como 0. Depois, fazemos a primeira chamada da função recursiva com o vértice inicial v_0 .

```

time = 0 # Global
def DFS(Grafo,v_inicial):
    """Faz uma busca em profundidade em um grafo"""
    G = Grafo # Para nao alterar os atributos originais do grafo

    nx.set_node_attributes(G,0,'d') # Tempo de entrada no vertice v
    nx.set_node_attributes(G,0,'f') # Tempo de saida no vertice v
    nx.set_node_attributes(G,'white','color') # white == v ainda nao visitado
                                                # gray == v ja visitado
                                                # black == todos os vizinhos de v ja foram visitados
    nx.set_node_attributes(G,None,'pi') # Predecessor de v

    global time
    time = 0

    DFS_visit(G,str(v_inicial)) # Chamada da funcao recursiva

```

Ao visitar um novo vértice, o contador de tempo é incrementado, atualiza-se o valor de d do vértice com o valor atual do tempo e a cor muda para cinza. Depois, visita-se um vizinho desse vértice. O valor de f de um vértice só será atualizado quando a chamada recursiva voltar, ou seja, todos os vizinhos dos vizinhos dos vizinhos (...) do vértice já foram visitados. O funcionamento característico da pilha é responsável por esse comportamento do algoritmo.

```

def DFS_visit(G, u):
    """Funcao recursiva que visita todos os vizinhos de um vertice"""
    global time
    time += 1

    G.node[str(u)]['d'] = time # Tempo de entrada em v
    G.node[str(u)]['color'] = 'gray' # Marca u como visitado

    for v in nx.neighbors(G,str(u)):           # Itera sobre os vizinhos de u
        if G.node[str(v)]['color'] == 'white':   # Se v ainda nao foi visitado
            G.node[str(v)]['pi'] = str(u)
            DFS_visit(G,str(v))

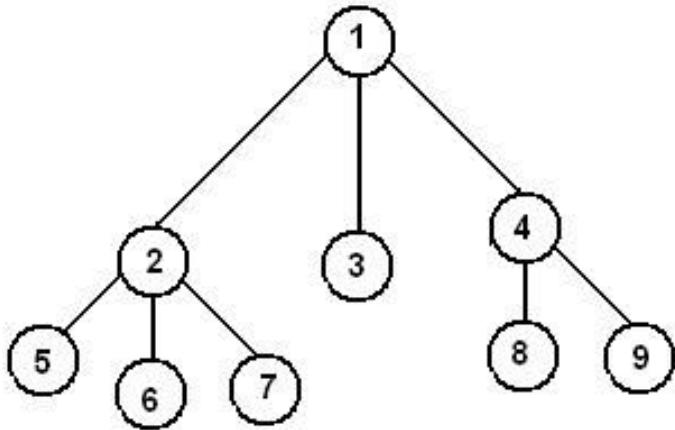
    G.node[str(u)]['color'] = 'black' # Marca u como 'toda vizinhança ja foi visitada'

    time += 1

    G.node[str(u)]['f'] = time # Tempo de saida em v

```

Como conclusão, é possível analisar como os algoritmos funcionam em um exemplo:



Ordem de visita com uma busca em largura:

[1, 2, 3, 4, 5, 6, 7, 8, 9]

Ordem de visita com uma busca em profundidade:

[1, 2, 5, 6, 7, 3, 4, 8, 9]

Problema 4:

Caminhos mínimos

Dado um Grafo $G = (V, E)$ e um conjunto de vértices iniciais $S \subseteq V$, obter a árvore de custo mínimo que conecta o vértices iniciais a todos os outros vértices, ou seja os caminhos do vértices iniciais para os outros vértices é o de menor custo, entretanto caso $|S| > 1$ ocorre uma disputa entre os vértices iniciais pelos demais, gerando assim uma floresta - *forest* - um grafo que é um conjunto de árvores, cada uma tendo um vértice inicial dado como origem. Seja v um dos vértices iniciais, $u \in V$ e $u \neq v$ e $P(u, v)$ o caminho de u a v na floresta gerada, então $\sum w(e), e \in P$, é mínimo. É feita uma busca M:N no grafo de entrada. Foi implementado para solucionar tal problema o algoritmo baseado no algoritmo de Dijkstra para caminhos mínimos e uma adaptação para uma *multisource*. Imagem da função Dijkstra_multisource:

```

def Dijkstra_multisource(G,S):
    nx.set_node_attributes(G, np.Inf, 'lambda') # setar lambda
    nx.set_node_attributes(G, None, 'pi') # setar pi

    lamb = {i: np.Inf for i in G.nodes().keys()} #dicionário com o lambda dos vertices
    lamb.update({i: 0 for i in S}) #o lambda de todos os vértices iniciais recebem 0
    for v in S:
        G.node[v]['lambda'] = 0

    while bool(lamb): #enquanto houver vértices no dicionário
        u = min(lamb, key=lamb.get) #pega o vértice com menor lambda

        for v in nx.neighbors(G,u): #para todos os vizinhos de u

            #se v estiver no dicionário e o custo para chegar em v
            #passando por u for menor que o lambda de v
            #atualização do lambda de v
            if v in lamb.keys() and G.edges[v,u]['weight'] + lamb[u] < lamb[v]:
                lamb[v] = G.edges[v,u]['weight'] + lamb[u]
                G.node[v]['lambda'] = G.edges[v,u]['weight'] + lamb[u]
                G.node[v]['pi'] = u #atualização do pi de v

        del lamb[u] #remover o vértice u

    resultados = {} #dicionario com os resultados
    resultados['pi'] = nx.get_node_attributes(G,'pi')
    resultados['lambda'] = nx.get_node_attributes(G,'lambda')

    Dm = pd.DataFrame(resultados) #Dataframe com os resultados

    Forest = nx.Graph() #criação da floresta
    Forest.add_nodes_from(G) #adição dos vértices de G na floresta

    for k in G.nodes().keys(): #Criação das arestas na floresta
        if resultados['pi'][k] != None: #se não for vértice inicial liga ao pai
            Forest.add_edge(k,resultados['pi'][k])

    return(Dm,Forest) #retorna uma tupla com o Dataframe e a floresta

```

Antes da chamada da função é feita uma leitura do conjunto de vértices iniciais que serão origem das árvores na floresta:

```

S = input() #leitura de uma linha com os vértices
S = S.split(' ')
S = [int(i) for i in S] #conjunto vértices iniciais

```

Inicialmente no algoritmo alguns atributos são criados para cada vértice do grafo, o lambda e o pi - λ e π respectivamente. O λ de um vértice diz respeito ao custo de sair de um vértice inicial e chegar até o respectivo vértice. O π representa o “pai” do vértice, se $\pi(u) = v$, então o vértice v é pai do vértice u , ou seja, existe aresta (u,v) na floresta. Para que seja mais fácil manipular os lambdas - que são mutáveis durante a execução algoritmo - e para não destruir o grafo de entrada é criado um dicionário $lamb$ com os valores de lambda de todos os vértices, que são inicializados com infinito. Os λ dos vértices iniciais são atualizados para 0. Código da parte descrita:

```

nx.set_node_attributes(G, np.Inf, 'lambda') # setar lambda
nx.set_node_attributes(G, None, 'pi') # setar pi

lamb = {i: np.Inf for i in G.nodes().keys()} #dicionário com o lambda dos vertices

lamb.update({i: 0 for i in S}) #o lambda de todos os vértices iniciais recebem 0
for v in S:
    G.node[v]['lambda'] = 0

```

Prosseguindo no algoritmo, começa o laço principal: enquanto houver vértice no dicionário, busque o vértice com o menor valor de lambda e para cada vizinho v do vértice u escolhido aplicar o relaxamento de aresta - se o custo de chegar em v passando por u for menor que o custo corrente de chegar em v , então atualizar o valor de $\lambda(v)$ e $\pi(v)$. Deletar o vértice u do dicionário após passar por todos os seus vizinhos:

```

while bool(lamb): #enquanto houver vértices no dicionário
    u = min(lamb, key=lamb.get) #pega o vértice com menor lambda

    for v in nx.neighbors(G,u): #para todos os vizinhos de u

        #se v estiver no dicionário e o custo para chegar em v
        #passando por u for menor que o lambda de v
        #atualização do lambda de v
        if v in lamb.keys() and G.edges[v,u]['weight'] + lamb[u] < lamb[v]:
            lamb[v] = G.edges[v,u]['weight'] + lamb[u]
            G.node[v]['lambda'] = G.edges[v,u]['weight'] + lamb[u]
            G.node[v]['pi'] = u #atualização do pi de v

    del lamb[u] #remover o vértice u

```

Ao final do algoritmo é criado um dicionário com os λ e π de todos os vértices que depois é transformado em um Dataframe. É criado a floresta - *Forest* - e adicionada a ela os vértices de G . Há um laço para cada vértice de G que adiciona uma aresta entre um vértice e seu pai - $\pi(v)$ caso não seja um dos vértices iniciais - não possuem pai. Retorna uma tupla contendo o Dataframe e a floresta *Forest*:

```

resultados = {} #dicionario com os resultados
resultados['pi'] = nx.get_node_attributes(G,'pi')
resultados['lambda'] = nx.get_node_attributes(G,'lambda')

Dm = pd.DataFrame(resultados) #Dataframe com os resultados

Forest = nx.Graph() #criação da floresta
Forest.add_nodes_from(G) #adição dos vértices de G na floresta

for k in G.nodes().keys(): #Criacao das arestas na floresta
    if resultados['pi'][k] != None: #se nao for vértice inicial liga ao pai
        Forest.add_edge(k,resultados['pi'][k])

return(Dm,Forest) #retorna uma tupla com o Dataframe e a floresta

```

Problema 5:

Problema do Caixeiro Viajante

Dado um grafo $G = (V, E, w)$ Hamiltoniano, obter um ciclo Hamiltoniano de custo mínimo, ou seja, $w(C) = \sum_{e \in C} w(e)$. Em outras palavras, busca-se um ciclo C contendo vértices do grafo G o qual a soma dos pesos das arestas de C seja o menor possível. No entanto, não se conhece uma solução ótima para o problema, apenas soluções subótimas (que se aproximam da solução ótima). Neste caso, foi implementado o algoritmo 2-Optimal. Imagem da função two_opt:

```

def two_opt(G,C):
    # As variáveis step1, step2, step3, step4 são para controlar os loops do algoritmo
    step1 = True
    while(step1):
        step1 = False
        global i
        global j
        i = 0
        step2 = True
        while(step2):
            step2 = False
            j = i + 2
            step3 = True
            while(step3):
                step3 = False
                Cij = C[:,]
                # Torção dos vértices
                swap = Cij[j]
                Cij[j] = Cij[i+1]
                Cij[i+1] = swap
                if(w(G, Cij) < w(G, C)): # Caso o novo caminho for menor, recomeçar o algoritmo usando o novo caminho
                    C = Cij[:,]
                    step1 = True
                    step4 = False
                else:
                    step4 = True
                    step1 = False
                while(step4): # Se não, avance o índice j
                    step4 = False
                    j = j + 1
                    n = nx.number_of_nodes(G)
                    if(j < n):
                        step3 = True # Faça nova torção
                    else: # Caso o índice j for maior ou igual ao número de vértices, avance i e recalcula j
                        i=i+1
                        if(i < n-2):
                            step2 = True
                        else: # Caso o i atingir a metade dos vértices, garantimos que C é o melhor caminho aproximado
                            return C

```

Função $w(G, \text{path})$:

```

# Essa função calcula o peso total do caminho
def w(G, path):
    sum = 0
    for i in range(0,len(path) - 1):
        sum = sum + G[path[i]][path[i+1]]['weight']
    return sum

```

MATERIAIS E MÉTODOS

Todos os algoritmos foram desenvolvidos em Python 3 - 3.6.6 - utilizando o ambiente Jupyter Notebook³. As bibliotecas do python utilizadas foram: Numpy, NetworkX e Pandas. Os códigos foram reproduzidos em Linux Mint e MacOS.

Dados utilizados nos problemas:

Problema 3:

Zachary's karate club: <http://networkdata.ics.uci.edu/data/karate/>

Dolphins social network: <http://networkdata.ics.uci.edu/data/dolphins/>

Problema 2 e 5:

Grafo com 30 vértices:

https://ava.ead.ufscar.br/pluginfile.php/599996/mod_label/intro/ha30.zip

Problema 4:

Grafo com 59 cidades da Alemanha ocidental e suas distâncias:

https://ava.ead.ufscar.br/pluginfile.php/599996/mod_label/intro/WG59.zip

RESULTADOS E DISCUSSÕES

Nesta seção há o resultado de todos os problemas, plotagem dos dados obtidos, como matrizes, grafos e caminhos.

Snakes and Ladders

Obtendo a Matriz P e sua distribuição estacionária:

```
In [6]: 1 P = delta @ adj_matrix                                     #Gera a matriz P
2 P

Out[6]: array([[0., 0.5, 0.5, ..., 0., 0., 0.],
   [0., 0., 0., ..., 0., 0., 0.],
   [0., 0., 0., ..., 0., 0., 0.],
   ...,
   [0., 0., 0., ..., 0., 0., 0.],
   [0., 0., 0., ..., 0., 0., 1.],
   [0., 0., 0., ..., 0., 0., 1.]])
```

Para $k = 100$, obtém-se a seguinte distribuição estacionária:

```
In [8]: 1 # Distribuicao Estacionaria de P
2 distr_estac_P = power_method(P,w,100)
3 distr_estac_P

Out[8]: array([0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.98761390e-03,
   1.02459294e-03, 1.02459294e-03, 1.58449826e-03, 1.34495718e-03,
   1.51010175e-03, 6.93310659e-04, 3.57394211e-04, 2.45306526e-03,
   1.44876097e-03, 2.01134833e-03, 1.78364802e-03, 1.95627878e-03,
   1.92789133e-03, 1.00844036e-03, 0.00000000e+00, 0.00000000e+00,
   0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
   0.00000000e+00, 0.00000000e+00, 1.55688234e-03, 8.02556150e-04,
   2.25594496e-03, 1.12788829e-02, 6.97705430e-03, 9.41073412e-03,
   3.59659439e-03, 1.85400481e-03, 1.85400481e-03, 9.38296846e-01])
```



```
In [22]: 1 print("Vertice com a maior probabilidade de ser atingido:", distr_estac_P.argmax())
2 print("Probabilidade de vencer o jogo:", distr_estac_P.max())
3 print("Soma de todas as probabilidades:", distr_estac_P.sum())

Vertice com a maior probabilidade de ser atingido: 35
Probabilidade de vencer o jogo: 0.9382968462656478
Soma de todas as probabilidades: 1.0
```

Como é possível observar, o vértice com a maior probabilidade de ser encontrado é o vértice 35, que corresponde à casa 36 do tabuleiro (casa final). Além disso, a probabilidade do vértice 35 ser eventualmente visitado é de aproximadamente 94%.

Obtendo a Matriz \bar{P} e sua distribuição estacionária:

```
In [10]: 1 # Matriz P barra
          2 P_ = Google_matrix(P,0.1)
          3 P_
Out[10]: array([[0.00277778, 0.45277778, 0.45277778, ..., 0.00277778, 0.00277778,
                 0.00277778],
                [0.00277778, 0.00277778, 0.00277778, ..., 0.00277778, 0.00277778,
                 0.00277778],
                [0.00277778, 0.00277778, 0.00277778, ..., 0.00277778, 0.00277778,
                 0.00277778],
                ...,
                [0.00277778, 0.00277778, 0.00277778, ..., 0.00277778, 0.00277778,
                 0.00277778],
                [0.00277778, 0.00277778, 0.00277778, ..., 0.00277778, 0.00277778,
                 0.90277778],
                [0.00277778, 0.00277778, 0.00277778, ..., 0.00277778, 0.00277778,
                 0.90277778]])
```

Utilizando novamente o Power Method, obtemos a seguinte distribuição estacionária.

```
In [11]: 1 # Distribuicao Estacionaria de P_
          2 distr_estac_P_ = power_method(P_,w,100)
          3 distr_estac_P_
Out[11]: array([0.00277778, 0.00402778, 0.00402778, 0.03073341, 0.01842031,
                 0.02023281, 0.02846083, 0.02468992, 0.02669562, 0.01388824,
                 0.00902749, 0.02797841, 0.01943043, 0.02411176, 0.02599677,
                 0.03238132, 0.02904792, 0.01734937, 0.00277778, 0.00402778,
                 0.00402778, 0.00459028, 0.0066559 , 0.00783856, 0.00577293,
                 0.00277778, 0.02805383, 0.016652 , 0.03850984, 0.09265671,
                 0.06180274, 0.07228454, 0.03058901, 0.01654284, 0.02173848,
                 0.22342326])
```

Como é possível observar, não há mais estados com probabilidade 0 de serem acessados, como era na matriz P .

```
In [23]: 1 distr_estac_P_.sum()
Out[23]: 0.9999999999999974
```

Obs: Por uma questão de cálculos computacionais, o somatório da matriz não resultou exatamente em 1, mas bem próximo disso

Comparando as matrizes:

```
In [13]: 1 distr_estac_P == distr_estac_P_
Out[13]: array([False, False, False, False, False, False, False, False,
   False, False, False, False, False, False, False, False, False,
   False, False, False, False, False, False, False, False, False,
   False, False, False, False, False, False, False, False])

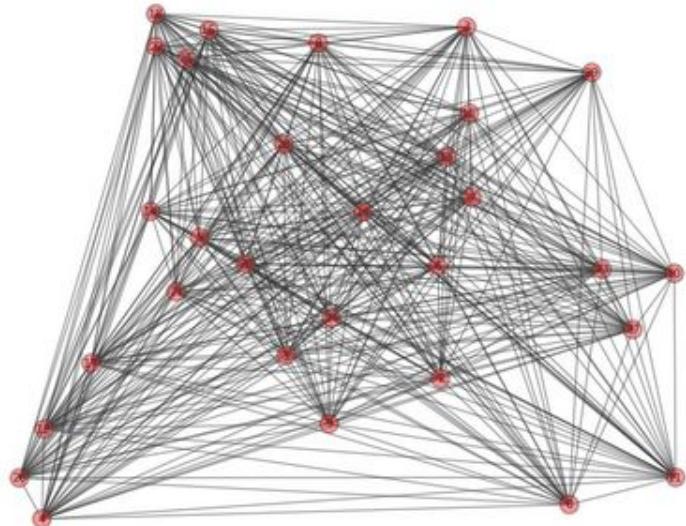
In [14]: 1 distr_estac_P_ > distr_estac_P
Out[14]: array([ True,  True,  True,  True,  True,  True,  True,  True,
   True,  True,  True,  True,  True,  True,  True,  True,  True,
   True,  True,  True,  True,  True,  True,  True,  True,  True,
   True,  True,  True,  True,  True,  True,  True,  True, False])

In [32]: 1 print("Vertice com a maior probabilidade de ser atingido:", distr_estac_P_.argmax())
2 print("Probabilidade de vencer o jogo:", distr_estac_P_.max())
3 print("Soma de todas as probabilidades:", distr_estac_P_.sum())
Vertice com a maior probabilidade de ser atingido: 35
Probabilidade de vencer o jogo: 0.22342326218710654
Soma de todas as probabilidades: 0.9999999999999974
```

Comparando as matrizes P e \bar{P} , percebe-se que a probabilidade de transição aumentou para todos os estados, menos para o estado final, já que agora, o jogador não fica mais "preso" no estado final ao atingi-lo.

Árvore Geradora Mínima

Foi instanciado um grafo G a partir de uma matriz A - *ha30_dist.txt* - que contém 30 cidades icônicas do mundo e o custo de viajar de uma até outra:



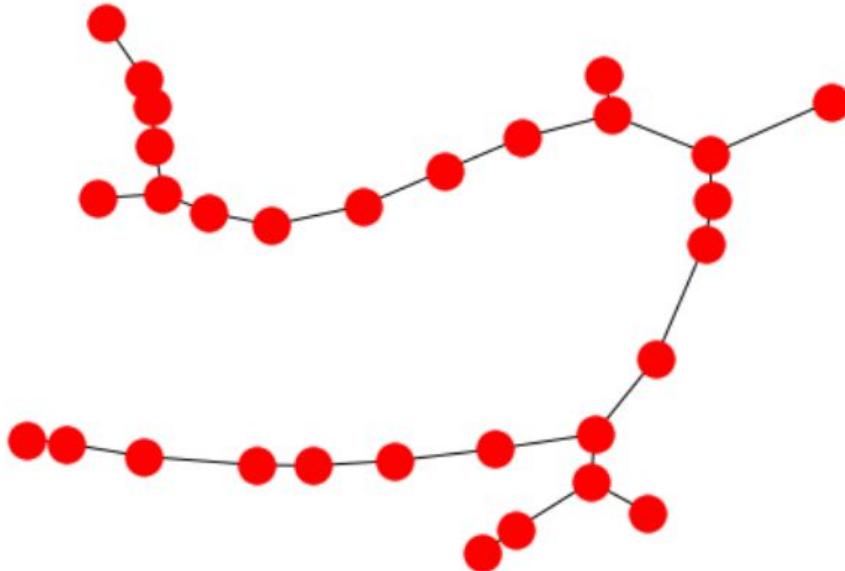
O código para a construção do grafo:

```
A = np.loadtxt('ha30/ha30_dist.txt')
G = nx.from_numpy_matrix(A)
```

Após a aquisição dos dados, é chamada a função e atribuída a uma tupla - TabResultados e T1. TabResultados é um Dataframe com os lambdas e pis finais de cada vértice, entretanto, para este problema o lambda final não tem muito significado - em outros algoritmos, como BFS(Busca em largura) e Dijkstra(Caminhos mínimos), há um valor e uma ideia mais concreta sobre ele. T1 é a árvore geradora mínima. Na implementação, há uma verificação - `nx.is_tree(T1)` - para só plotar o grafo se ele for uma árvore:

```
(TabResultados,T1) = MST_Prim(G,1)
if nx.is_tree(T1):
    nx.draw(T1)
    plt.show() # display
```

Abaixo o grafo T1 resultante com o vértice inicial sendo o 1:



E também a tabela com o lambda e os pi de cada vértice:

	pi	lambda		pi	lambda
0	21.0	16.0	14	28.0	4.0
1	NaN	0.0	15	18.0	9.0
2	23.0	7.0	16	0.0	24.0
3	1.0	20.0	17	2.0	10.0
4	25.0	7.0	18	7.0	8.0
5	1.0	8.0	19	16.0	3.0
6	22.0	38.0	20	15.0	15.0
7	19.0	7.0	21	2.0	5.0
8	29.0	16.0	22	4.0	12.0
9	24.0	24.0	23	10.0	9.0
10	5.0	8.0	24	26.0	7.0
11	26.0	9.0	25	20.0	31.0
12	21.0	2.0	26	7.0	17.0
13	27.0	12.0	27	3.0	31.0
28	8.0	33.0	29	27.0	11.0

Além disso, outra métrica importante é o custo mínimo da árvore obtida, para isso foi realizada uma soma de todas as arestas e seus respectivos pesos:

```
In [5]: s = 0
for e in T1.edges:
    s += T1.edges[e[0],e[1]]['weight']
print(s)
```

403.0

Para qualquer que seja o vértice inicial a soma dos vértices da árvore gerada sempre será a mesma, uma vez que só há uma árvore geradora mínima.

Problema 3: Busca em Grafos

Os resultados para busca em largura e busca em profundidade tanto para o grafo Dolphins quanto para o grafo Karate são expressos tanto pela árvore gerada quanto pela tabela de atributos de vértices ao final da execução do algoritmo.

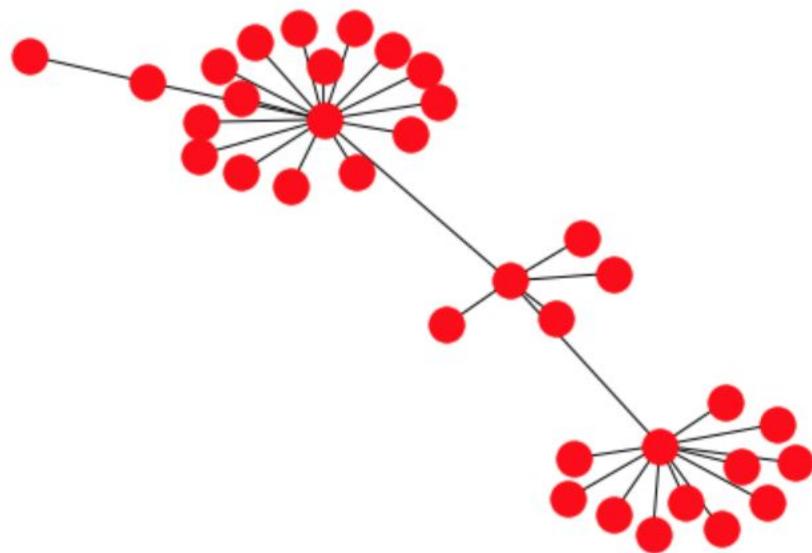
Obs: Foi utilizada uma função da biblioteca NetworkX chamada `is_tree` para garantir que o retorno do algoritmo é uma árvore.

Todos esses resultados estão expostos abaixo:

Busca em Largura:

Karate:

```
In [6]: 1 (bfs_karate,T2) = BFS(G_karate,1)
2 if nx.is_tree(T2):
3     nx.draw(T2)
4     plt.show()
```



Árvore BFS gerada

In [10]: 1 bfs_karate

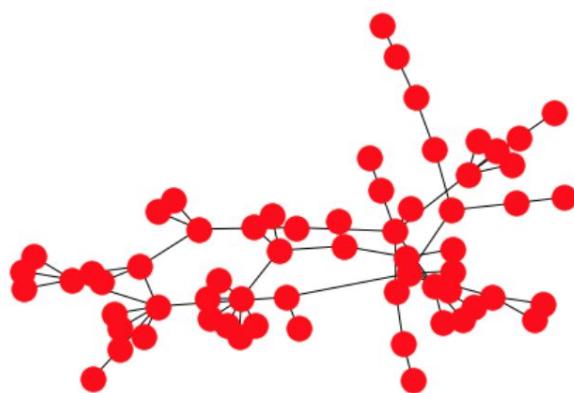
Out[10]:

	pi	lambda	color
1	None	0	black
10	34	3	black
11	1	1	black
12	1	1	black
13	1	1	black
14	1	1	black
15	34	3	black
16	34	3	black
17	7	2	black
18	1	1	black
19	34	3	black
2	1	1	black
20	1	1	black
21	34	3	black
22	1	1	black
23	34	3	black
24	34	3	black
25	32	2	black
26	32	2	black
27	34	3	black
28	34	3	black
29	32	2	black
3	1	1	black

Atributos finais dos vértices

Dolphins:

```
In [8]: 1 (bfs_dolphins,T4) = BFS(G_dolphins,0)
2 if nx.is_tree(T4):
3     nx.draw(T4)
4     plt.show()
```



Árvore BFS gerada

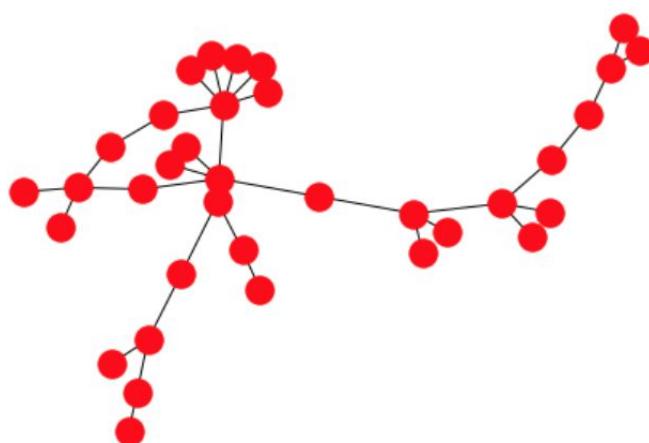
	In [12]:	1	bfs_dolphins
	Out[12]:		
		pi	lambda
0	None	0	black
1	19	4	black
10	0	1	black
11	51	11	black
12	33	12	black
13	54	5	black
14	0	1	black
15	0	1	black
16	50	11	black
17	57	6	black
18	45	10	black
19	30	3	black
2	61	15	black
20	47	2	black
21	45	10	black
22	17	7	black
23	36	8	black
24	45	10	black
25	17	7	black
26	27	8	black
27	17	7	black
28	47	2	black
29	45	10	black

Atributos finais dos vértices

Busca em Profundidade:

Karate:

```
In [5]: 1 # Rodando o algoritmo para cada grafo
2 # Karate
3 (dfs_karate,T1) = DFS(G_karate,1)
4 if nx.is_tree(T1):
5     nx.draw(T1)
6     plt.show() # display
```



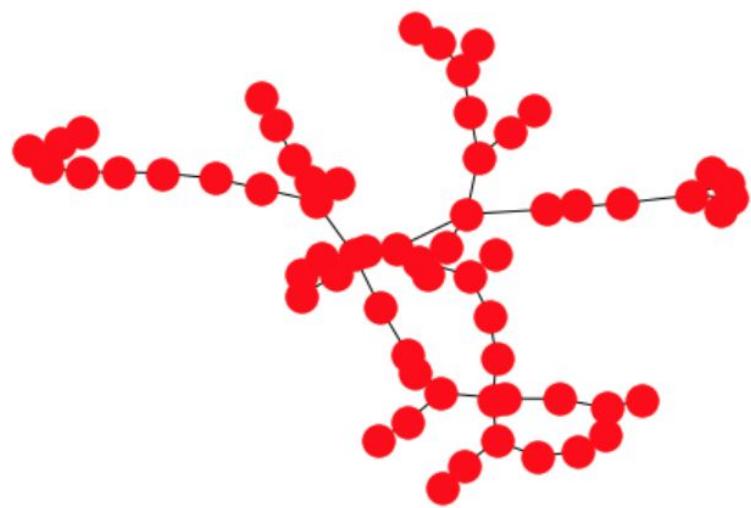
Árvore DFS gerada

In [9]:	1	dfs_karate
Out[9]:		
	1	d
	10	f
	11	color
	12	pi
	13	black
	14	None
	15	34
	16	6
	17	1
	18	4
	19	33
	20	2
	21	33
	22	1
	23	34
	24	33
	25	2
	26	33
	27	26
	28	24
	29	30
	30	25

Atributos finais dos vértices

Dolphins:

```
In [7]: 1 (dfs_dolphins,T3) = DFS(G_dolphins,0)
2 if nx.is_tree(T3):
3     nx.draw(T3)
4     plt.show()
```



Árvore DFS gerada

In [11]:	1	dfs_dolphins		
Out[11]:	d	f	color	pi
0	1	124	black	None
1	8	117	black	19
10	2	123	black	0
11	61	62	black	51
12	29	30	black	33
13	13	100	black	5
14	26	85	black	3
15	33	80	black	18
16	27	84	black	14
17	9	116	black	1
18	32	81	black	21
19	7	118	black	7
2	3	122	black	10
20	23	92	black	36
21	31	82	black	33
22	106	107	black	17
23	57	68	black	45
24	34	79	black	15
25	108	113	black	17
26	109	112	black	25
27	110	111	black	26
28	87	90	black	8
29	35	78	black	24

Atributos finais dos vértices

Obs: Nenhuma das tabelas acima contém todos os vértices. As tabelas eram muito grandes e somente uma parte delas foi exibida aqui.

Caminhos mínimos

Inicialmente há um carregamento de uma matriz - *wg59_dist.txt* - com os valores dos pesos das arestas existentes e a construção do grafo a partir dela:

```
A = np.loadtxt('WG59/wg59_dist.txt')
G = nx.from_numpy_matrix(A)
```

Chamada da função que retorna uma tupla, um dataframe - *r* - com os valores de lambda - custo para se chegar no vértice a partir de um dos vértices iniciais - e *pi* de cada vértice, e a floresta *F*:

```
(r,F) = Dijkstra_multisource(G,S)
if nx.is_forest(F):
    nx.draw(F)
    plt.show() # display
```

Obs: Foi utilizada uma função da biblioteca NetworkX chamada `is_forest` para garantir que o retorno do algoritmo é uma floresta.

Floresta gerada a partir de dois vértices iniciais, 2 e 34:



Tabela com os valores de lambda e pi:

	pi	lambda		pi	lambda		pi	lambda
0	23.0	57.0	10	34.0	60.0	20	34.0	66.0
1	2.0	35.0	11	2.0	79.0	21	2.0	20.0
2	NaN	0.0	12	34.0	78.0	22	2.0	15.0
3	2.0	69.0	13	2.0	52.0	23	34.0	18.0
4	34.0	64.0	14	23.0	85.0	24	2.0	66.0
5	2.0	5.0	15	2.0	43.0	25	34.0	37.0
6	34.0	110.0	16	2.0	19.0	26	2.0	115.0
7	2.0	45.0	17	34.0	53.0	27	34.0	67.0
8	21.0	99.0	18	34.0	9.0	28	34.0	66.0
9	1.0	65.0	19	47.0	59.0	29	34.0	34.0
30	2.0	36.0	40	2.0	12.0	50	34.0	40.0
31	34.0	29.0	41	2.0	12.0	51	55.0	121.0
32	34.0	58.0	42	34.0	15.0	52	34.0	71.0
33	34.0	11.0	43	2.0	61.0	53	2.0	41.0
34	NaN	0.0	44	18.0	45.0	54	34.0	96.0
35	2.0	28.0	45	34.0	19.0	55	2.0	35.0
36	2.0	81.0	46	2.0	4.0	56	34.0	72.0
37	2.0	74.0	47	2.0	18.0	57	2.0	81.0
38	18.0	14.0	48	2.0	7.0	58	2.0	29.0
39	2.0	10.0	49	34.0	18.0			

Outro resultado, floresta gerada a partir de três vértices iniciais, 5,10 e 47:

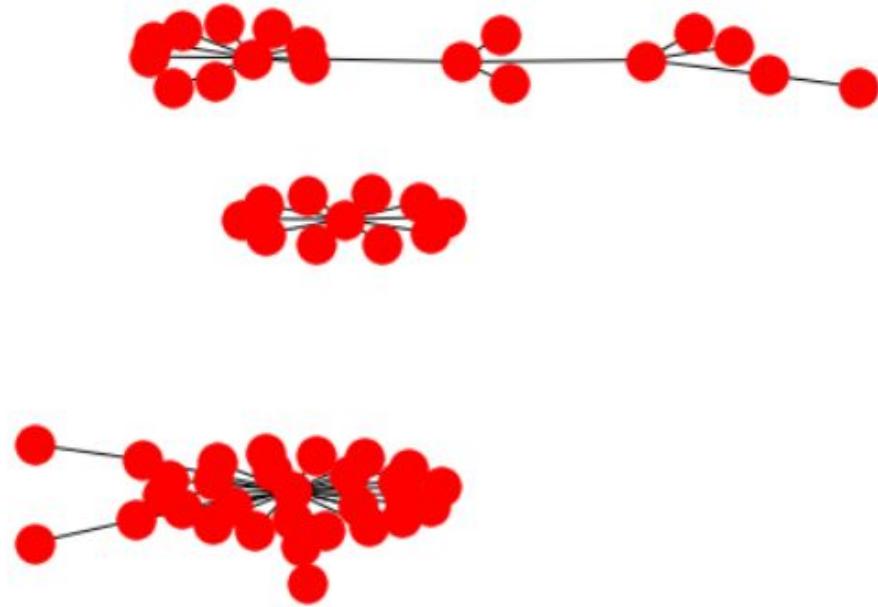


Tabela com o lambda e pi de cada vértice:

	pi	lambda		pi	lambda		pi	lambda
0	10.0	49.0	10	NaN	0.0	20	0.0	67.0
1	5.0	40.0	11	10.0	35.0	21	46.0	23.0
2	5.0	5.0	12	10.0	18.0	22	5.0	10.0
3	21.0	72.0	13	47.0	45.0	23	10.0	48.0
4	10.0	31.0	14	10.0	58.0	24	46.0	68.0
5	NaN	0.0	15	47.0	26.0	25	10.0	71.0
6	10.0	50.0	16	47.0	14.0	26	46.0	118.0
7	47.0	28.0	17	50.0	75.0	27	10.0	11.0
8	21.0	102.0	18	10.0	59.0	28	10.0	32.0
9	47.0	70.0	19	10.0	40.0	29	10.0	32.0

	pi	lambda		pi	lambda		pi	lambda
30	5.0	31.0	40	5.0	8.0	50	10.0	51.0
31	10.0	33.0	41	5.0	7.0	51	55.0	124.0
32	10.0	31.0	42	31.0	47.0	52	10.0	22.0
33	10.0	51.0	43	10.0	39.0	53	47.0	27.0
34	10.0	60.0	44	47.0	53.0	54	10.0	36.0
35	47.0	15.0	45	10.0	45.0	55	21.0	38.0
36	47.0	82.0	46	5.0	4.0	56	10.0	20.0
37	5.0	77.0	47	NaN	0.0	57	10.0	52.0
38	10.0	60.0	48	5.0	11.0	58	47.0	11.0
39	5.0	15.0	49	10.0	53.0			

Problema 5:

Problema do Caixeiro Viajante

Foi instanciado um grafo G a partir de uma matriz A - *ha30_dist.txt* - que representa 30 cidades icônicas do mundo e o custo de viajar de uma até outra.

Também foi carregado uma lista names - *ha30_name.txt* - contendo o nome das 30 cidades:

```
A = np.loadtxt('ha30/ha30_dist.txt')
G = nx.from_numpy_matrix(A)
names = open('ha30/ha30_name.txt', 'r').read().splitlines()
```

Posteriormente, foram realizadas 10 execuções do algoritmo 2-optimal, alterando a inicialização a cada execução. Não foi adotada nenhuma heurística para escolha do ciclo inicial, a escolha é feita de acordo com índice k que representa o número da execução.

```
mapa = {}
bests = []
# realizando 10 execuções
for k in range(0, 10):
    P = names[:]
    # alterando a inicialização a cada execução
    if(k>0):
        swap = P[0]
        P[0] = P[k+2]
        P[k+2] = swap
    # renomeando os vértices do grafo
    for i in range(0,30):
        mapa[i] = P[i]
    G = nx.relabel_nodes(G, mapa)

    best = two_opt(G,P) # obtendo o ciclo a partir do algoritmo 2-optimal
    print("k: " + str(k) + " w(C): " + str(w(G,best)))
    print(best)

    bests.append((w(G,best), best)) # criando lista dos ciclos e seu respectivo peso

# ordenando a lista
bests.sort(key=lambda x: x[0])
```

Resultado:

```
k: 0 w(C): 625.0
['Azores', 'New Orleans', 'Panama City', 'Santiago', 'Buenos Aires', 'Rio de Janeiro', 'Capetown', 'Rome', 'Istanbul', 'Cairo', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney', 'Guam', 'Honolulu', 'Mexico City', 'Chicago', 'New York', 'Montreal', 'London', 'Paris', 'Berlin', 'Moscow', 'Seattle', 'San Francisco', 'Juneau', 'Tokyo', 'Shanghai', 'Manila']
k: 1 w(C): 639.0
['Bombay', 'Moscow', 'Montreal', 'New York', 'Chicago', 'Seattle', 'Juneau', 'Azores', 'London', 'Paris', 'Berlin', ' Rome', 'Istanbul', 'Melbourne', 'Sydney', 'Honolulu', 'San Francisco', 'New Orleans', 'Mexico City', 'Panama City', 'Santiago', 'Buenos Aires', 'Rio de Janeiro', 'Capetown', 'Cairo', 'Baghdad', 'Manila', 'Shanghai', 'Tokyo', 'Guam']
k: 2 w(C): 702.0
['Buenos Aires', 'Capetown', 'Cairo', 'Bombay', 'Manila', 'Guam', 'Honolulu', 'Mexico City', 'San Francisco', 'Seattle', 'Juneau', 'Berlin', 'Moscow', 'Sydney', 'Melbourne', 'Santiago', 'Rio de Janeiro', 'Panama City', 'New Orleans', 'Chicago', 'New York', 'Montreal', 'Azores', 'London', 'Paris', 'Rome', 'Istanbul', 'Baghdad', 'Shanghai', 'Tokyo']
k: 3 w(C): 693.0
['Cairo', 'Istanbul', 'Berlin', 'London', 'Chicago', 'San Francisco', 'Seattle', 'Juneau', 'Tokyo', 'Shanghai', 'Moscow', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney', 'Mexico City', 'New Orleans', 'New York', 'Montreal', 'Azores', 'Paris', 'Rome', 'Capetown', 'Rio de Janeiro', 'Buenos Aires', 'Santiago', 'Panama City', 'Honolulu', 'Guam', 'Manila']
k: 4 w(C): 677.0
['Capetown', 'Cairo', 'Berlin', 'Montreal', 'San Francisco', 'Seattle', 'Juneau', 'Tokyo', 'Shanghai', 'Moscow', 'Istanbul', 'Baghdad', 'Bombay', 'Manila', 'Honolulu', 'Mexico City', 'Panama City', 'New Orleans', 'Chicago', 'New York', 'Azores', 'London', 'Paris', 'Rome', 'Rio de Janeiro', 'Buenos Aires', 'Santiago', 'Melbourne', 'Sydney', 'Guam']
k: 5 w(C): 704.0
['Chicago', 'Rome', 'Cairo', 'Rio de Janeiro', 'Santiago', 'Buenos Aires', 'Capetown', 'Manila', 'Guam', 'Honolulu', 'San Francisco', 'Seattle', 'Juneau', 'Tokyo', 'Shanghai', 'Montreal', 'New Orleans', 'Mexico City', 'Panama City', 'New York', 'Azores', 'London', 'Paris', 'Berlin', 'Moscow', 'Istanbul', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney']
k: 6 w(C): 677.0
['Guam', 'Capetown', 'Buenos Aires', 'Rio de Janeiro', 'Cairo', 'Istanbul', 'Moscow', 'Berlin', 'London', 'Juneau', 'Seattle', 'San Francisco', 'Honolulu', 'Santiago', 'Panama City', 'Mexico City', 'New Orleans', 'Chicago', 'New York', 'Montreal', 'Azores', 'Paris', 'Rome', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney', 'Manila', 'Shanghai', 'Tokyo']
k: 7 w(C): 704.0
['Honolulu', 'Moscow', 'Berlin', 'Istanbul', 'Cairo', 'Capetown', 'Rio de Janeiro', 'Buenos Aires', 'Santiago', 'Panama City', 'Rome', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney', 'Mexico City', 'New Orleans', 'Chicago', 'New York', 'Montreal', 'Azores', 'Paris', 'London', 'Seattle', 'San Francisco', 'Juneau', 'Guam', 'Manila', 'Shanghai', 'Tokyo']
k: 8 w(C): 645.0
['Istanbul', 'Baghdad', 'Cairo', 'Capetown', 'Buenos Aires', 'Santiago', 'Melbourne', 'Sydney', 'Manila', 'Bombay', 'Rome', 'London', 'Juneau', 'Seattle', 'San Francisco', 'Chicago', 'Montreal', 'New York', 'New Orleans', 'Mexico City', 'Panama City', 'Rio de Janeiro', 'Azores', 'Paris', 'Berlin', 'Moscow', 'Shanghai', 'Guam', 'Honolulu']
k: 9 w(C): 657.0
['Juneau', 'Moscow', 'Istanbul', 'Cairo', 'Baghdad', 'Bombay', 'Manila', 'Shanghai', 'Tokyo', 'Seattle', 'San Francisco', 'Chicago', 'London', 'Berlin', 'Rome', 'Paris', 'Panama City', 'Mexico City', 'New Orleans', 'New York', 'Montreal', 'Azores', 'Capetown', 'Rio de Janeiro', 'Buenos Aires', 'Santiago', 'Honolulu', 'Guam', 'Sydney', 'Melbourne']
```

Depois, foram selecionadas as 3 melhores soluções:

```
print("1: custo=" + str(bests[0][0]))
print(bests[0][1])
print("2: custo=" + str(bests[1][0]))
print(bests[1][1])
print("3: custo=" + str(bests[2][0]))
print(bests[2][1])

1: custo=625.0
['Azores', 'New Orleans', 'Panama City', 'Santiago', 'Buenos Aires', 'Rio de Janeiro', 'Capetown', 'Rome', 'Istanbul', 'Cairo', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney', 'Guam', 'Honolulu', 'Mexico City', 'Chicago', 'New York', 'Montreal', 'London', 'Paris', 'Berlin', 'Moscow', 'Seattle', 'San Francisco', 'Juneau', 'Tokyo', 'Shanghai', 'Manila']
2: custo=639.0
['Bombay', 'Moscow', 'Montreal', 'New York', 'Chicago', 'Seattle', 'Juneau', 'Azores', 'London', 'Paris', 'Berlin', ' Rome', 'Istanbul', 'Melbourne', 'Sydney', 'Honolulu', 'San Francisco', 'New Orleans', 'Mexico City', 'Panama City', 'Santiago', 'Buenos Aires', 'Rio de Janeiro', 'Capetown', 'Cairo', 'Baghdad', 'Manila', 'Shanghai', 'Tokyo', 'Guam']
3: custo=645.0
['Istanbul', 'Baghdad', 'Cairo', 'Capetown', 'Buenos Aires', 'Santiago', 'Melbourne', 'Sydney', 'Manila', 'Bombay', 'Rome', 'London', 'Juneau', 'Seattle', 'San Francisco', 'Chicago', 'Montreal', 'New York', 'New Orleans', 'Mexico City', 'Panama City', 'Rio de Janeiro', 'Azores', 'Paris', 'Berlin', 'Moscow', 'Shanghai', 'Tokyo', 'Guam', 'Honolulu']
```

E as 3 piores soluções:

```
print("8: custo=" + str(bests[7][0]))
print(bests[7][1])
print("9: custo=" + str(bests[8][0]))
print(bests[8][1])
print("10: custo=" + str(bests[9][0]))
print(bests[9][1])

8: custo=702.0
['Buenos Aires', 'Capetown', 'Cairo', 'Bombay', 'Manila', 'Guam', 'Honolulu', 'Mexico City', 'San Francisco', 'Seattle', 'Juneau', 'Berlin', 'Moscow', 'Sydney', 'Melbourne', 'Santiago', 'Rio de Janeiro', 'Panama City', 'New Orleans', 'Chicago', 'New York', 'Montreal', 'Azores', 'London', 'Paris', 'Rome', 'Istanbul', 'Baghdad', 'Shanghai', 'Tokyo']
9: custo=704.0
['Chicago', 'Rome', 'Cairo', 'Rio de Janeiro', 'Santiago', 'Buenos Aires', 'Capetown', 'Manila', 'Guam', 'Honolulu', 'San Francisco', 'Seattle', 'Juneau', 'Tokyo', 'Shanghai', 'Montreal', 'New Orleans', 'Mexico City', 'Panama City', 'New York', 'Azores', 'London', 'Paris', 'Berlin', 'Moscow', 'Istanbul', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney']
10: custo=704.0
['Honolulu', 'Moscow', 'Berlin', 'Istanbul', 'Cairo', 'Capetown', 'Rio de Janeiro', 'Buenos Aires', 'Santiago', 'Panama City', 'Rome', 'Baghdad', 'Bombay', 'Melbourne', 'Sydney', 'Mexico City', 'New Orleans', 'Chicago', 'New York', 'Montreal', 'Azores', 'Paris', 'London', 'Seattle', 'San Francisco', 'Juneau', 'Guam', 'Manila', 'Shanghai', 'Tokyo']
```

E por fim, obteve-se a diferença do custo entre a melhor e a pior solução:

```
print("Diferença: " + str(bests[9][0] - bests[0][0]))
```

Diferença: 79.0

CONSIDERAÇÕES FINAIS

O objetivo deste artigo é, de maneira simples e direta, introduzir o leitor ao campo de estudos da teoria dos grafos e enunciar 5 problemas interessantes e suas respectivas possíveis soluções computacionais. Além disso, outra proposta é abordar o manuseio e implementação da linguagem Python - versão 3.6.6 - possibilitando uma visão dos algoritmos de um ponto de vista prático e aplicável em problemas comuns. Outro objetivo é ilustrar de maneira básica - visto a dimensão dos problemas e a necessidade de mostrar a solução correta - os grafos gerados.

O primeiro problema, Snakes and Ladders, é visto como uma ponte entre teoria dos grafos e probabilidade e estatística, contribuindo para que seja possível interpretar o tabuleiro do jogo como sendo um grafo e a partir disso extrair dados interessantes do mesmo, como a probabilidade de se chegar ao final. Já para os problemas 2,3 e 4, o algoritmo de busca em largura - BFS, o próprio problema 3 - se mostrou muito relevante na teoria dos grafos, já que é a base dos outros dois problemas - Árvore geradora mínima por Prim e Caminhos mínimos por Dijkstra. Ou seja, a visão utilizada nesses problemas para as suas respectivas resoluções é muito interessante e elegante, apesar de simples. Logo, a simplicidade da solução e a importância dos problemas fizeram com que fossem abordados neste artigo. Outro problema que pode ser resolvido com busca em largura é descobrir se o grafo é bipartido ou não, caso especial de coloração de vértices.

O último problema, Problema do Caixeiro Viajante, é interessante pois este representa muitos problemas do campo da logística, além de ser o único problema NP-Hard neste artigo, e por mais que não tenha uma solução ótima, as soluções aproximadas entregam um resultado muito satisfatório, mesmo com entradas

grandes. Quando foi comparado o custo entre a melhor e pior solução, pôde-se perceber que a diferença foi relativamente baixa, mas não ao ponto de descartar múltiplas execuções para selecionar a melhor solução, principalmente quando a entrada é grande.

De maneira geral, as soluções obtidas experimentalmente eram as esperadas, entretanto é preciso levar em conta a forma como foi implementado as resoluções, onde tivemos suporte da linguagem. Quanto ao custo computacional, se levado em conta o tamanho dos problemas propostos, foi adequado, e a velocidade das execuções também foram rápidas de acordo com o tamanho. Talvez com problemas onde há milhares de vértices e arestas, o custo de memória seja relativamente maior e a velocidade diminua proporcionalmente.

É recomendado fortemente o uso da linguagem python para a elaboração dos problemas propostos e para outras implementações de algoritmos que utilizam grafos, uma vez que a biblioteca NetworkX facilita o manuseio, busca, atributos de vértice, vizinhança de vértice, lista de graus entre outras infinidades de propriedades dos grafos de maneira robusta e simples, o que outras linguagens não possuem. É possível que linguagens compiladas - C/C++ por exemplo - tenham um desempenho quanto a velocidade e uso de memória melhor que na linguagem Python, entretanto algumas funcionalidades essenciais de grafos terão de ser implementadas pelo programador - possivelmente bem complexas e trabalhosas, como plotar a imagem do grafo.

REFERÊNCIAS

CORMEN, T. et al. Introduction to Algorithms. 3 ed. MIT Press, 2009.

¹<https://python.org.br/>

²<https://networkx.github.io/documentation/networkx-1.10/tutorial/tutorial.html>

³<http://jupyter.org/>