Tutorial Projeto 3
Alunos:

Rafael Bastos Saito - 726580
Renata Sarmet Smiderle Mendes - 726586
Rodrigo Pesse de Abreu - 726588

Busca em Largura e Profundidade

Um algoritmo de busca é um algoritmo que percorre um grafo andando pelos arcos de um vértice a outro. Um algoritmo de busca examina sistematicamente os vértices e os arcos do grafo; depois de examinar a ponta inicial de um arco, o algoritmo percorre o arco e examina sua ponta final. Cada arco é examinado no máximo uma vez.

Há muitas maneiras de organizar uma busca. Cada estratégia de busca é caracterizada pela ordem em que os vértices são examinados. Este tutorial introduz a busca em largura (BFS) e a busca em profundidade (DFS).

Busca em Largura (BFS)

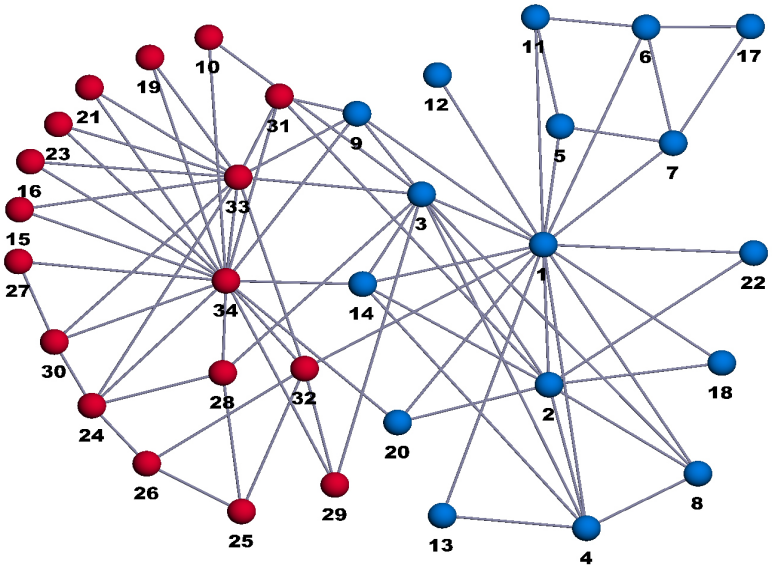
A busca em largura começa por um vértice, digamos s. O algoritmo visita s, depois visita todos os vizinhos de s, depois todos os vértices que estão à distância 2 de s, e assim por diante. Para implementar essa ideia, o algoritmo usa uma fila de vértices.

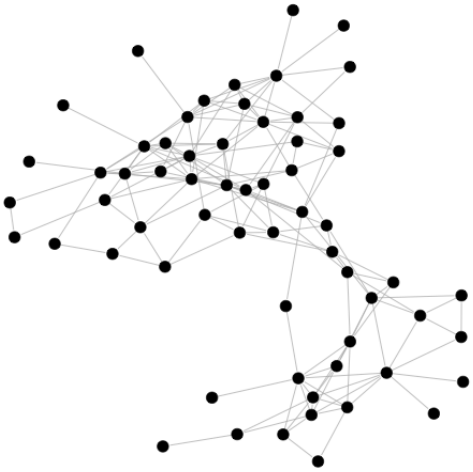
Busca em Profundidade (DFS)

A busca em profundidade começa por um vértice, digamos s. O algoritmo visita s, depois um vizinho de s, colocando os outros na pilha, depois um vizinho t de s, colocando os outros na pilha, depois um vizinho u de t e assim por diante, até esgotar os vizinhos não visitados, retornando para os próximos da pilha. Para implementar essa ideia, o algoritmo usa uma pilha de vértices.

Projeto

Implementar os algoritmos BFS e DFS para extrair as árvores BFS-tree e DFS-tree dos grafos a seguir.





Metodologia

Para a implementação foi utilizada a linguagem Python, a biblioteca NetworkX e Matplotlib para desenhar o grafo.

BFS.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as n

# Algoritmo de Busca em Largura
def BFS(G, s):
    cor  = {}                           # Cores indicando disponivel/ na fila/ completado
    pred = {}                           # Predecessores
    d    = {}                           # Distância

    for v in G.nodes():                 # Para todos os nós do grafo G
        d[v]    = n.inf                 # Inicializa todas as distâncias com infinito
        cor[v]  = 'branco'              # Branco, cinza e preto. Inicializa todos com branco (disponíveis)
        pred[v] = None                  # Inicializa todos os predecessores com nulo

    cor[s]  = 'cinza'                   # Cinza = na fila
    d[s]    = 0                         # Nó de origem tem distância zero

    Q = [ s ]                           # Colocado na fila

    while Q:                            # Enquanto a fila não estiver vazia
        u = Q.pop(0)                    # Retira o primeiro da fila
        for v in G.neighbors(u):        # Para todos os vizinhos de u
            if cor[v] == 'branco':      # Se o nó ainda estiver disponível
                cor[v]  = 'cinza'       # Muda a cor para indicar que está na fila
                d[v]    = d[u] + 1      # Atualiza a distância dele para a distância de u + 1
                pred[v] = u             # Coloca u como predecessor

                Q.append(v)             # Adiciona na fila

        cor[u] = 'preto'                # Depois de checado todos os vizinhos de u, marca ele como completado

    H = nx.create_empty_copy(G)         # Cria uma cópia de G com todas as arestas removidas

    for v1,v2 in G.edges():             # Retorna as arestas de v1 e v2 presentes em G, com default data {}

        #Se o predecessor de v2 for v1 ou se o predecessor de v1 for v2 e não for um grafo direcional
        if (pred[v2] is v1) or (pred[v1] is v2 and not nx.is_directed(H)):
            H.add_edge(v1, v2)              # Adiciona essa aresta em H
            H.node[v1]['depth'] = d[v1]     # Adiciona o atributo profundidade em v1 com a distância salva dele
            H.node[v2]['depth'] = d[v2]     # Adiciona o atributo profundidade em v2 com a distância salva dele

    return H                            # Retorna o grafo H com a BFS-tree
```

Definindo BFS

DFS.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as n

# Criando estrutura de pilha para ser utilizada
class Pilha(object):
    def __init__(self):
        self.dados = []

    def empilha(self, elemento):                    # Método para empilhar
        self.dados.append(elemento)                 # Adiciona no final do vetor de dados

    def desempilha(self):                           # Método para desempilhar
        if not self.vazia():                        # Verifica se a pilha não está vazia
            return self.dados.pop(-1)               # Se não estiver, retira o último elemento do vetor (último a entrar)

    def vazia(self):                                # Método para verificar se a pilha está vazia
        return len(self.dados) == 0                 # Se o tamanho do vetor de dados for zero, então está vazia
```

Definindo pilha em DFS

DFS.py

```python
# Algoritmo de Busca em Profundidade
def DFS(G, s):
    cor  = {}                                       # Cores indicando disponivel/ na pilha/ completado
    pred = {}                                       # Predecessores
    d    = {}                                       # Distância

    for v in G.nodes():                             # Para todos os nós do grafo G
        d[v]    = n.inf                             # Inicializa todas as distâncias com infinito
        cor[v]  = 'branco'                          # Branco, cinza e preto. Inicializa todos com branco (disponíveis)
        pred[v] = None                              # Inicializa todos os predecessores com nulo

    cor[s]  = 'cinza'                               # Cinza = na pilha

    p = Pilha()                                     # Cria Pilha p
    p.empilha(s)                                    # Colocado na pilha

    while not p.vazia():                            # Enquanto a pilha não estiver vazia
        u = p.desempilha()                          # Desempilha
        if pred[u] == None:                         # Se for o primeiro nó (raiz)
            d[u] = 0                                # Inicializa distância com 0
        else:                                       # Se não, se for qualquer outro nó
            d[u] = d[pred[u]] + 1                   # Atualiza a distância dele para a distância do predecessor + 1

        for v in G.neighbors(u):                    # Para todos os vizinhos de u
            if cor[v] == 'branco' or cor[v] == 'cinza':  # Se o nó ainda estiver disponível ou na pilha
                cor[v]  = 'cinza'                   # Garante que a cor indicará que está na pilha
                pred[v] = u                         # Coloca u como predecessor

                p.empilha(v)                        # Adiciona na pilha

        cor[u] = 'preto'                            # Depois de checado todos os vizinhos de u, marca ele como completado

    H = nx.create_empty_copy(G)                     # Cria uma cópia de G com todas as arestas removidas

    for v1,v2 in G.edges():                         # Retorna as arestas de v1 e v2 presentes em G, com default data {}

        #Se o predecessor de v2 for v1 ou se o predecessor de v1 for v2 e não for um grafo direcional
        if (pred[v2] is v1) or (pred[v1] is v2 and not nx.is_directed(H)):

            H.add_edge(v1, v2)                      # Adiciona essa aresta em H
            H.node[v1]['depth'] = d[v1]             # Adiciona o atributo profundidade em v1 com a distância salva dele
            H.node[v2]['depth'] = d[v2]             # Adiciona o atributo profundidade em v2 com a distância salva dele

    return H                                        # Retorna o grafo H com a BFS-tree
```

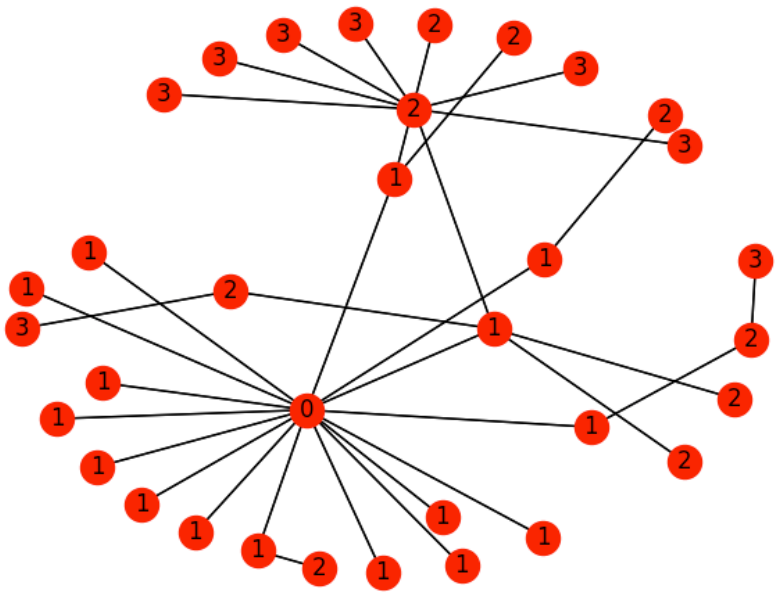
Definindo DFS

Foi rodado a busca em largura e profundidade para Zachary's karate club.

testaKarateBFS.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as n
import matplotlib.pyplot as plt
from BFS import BFS                    # Importa método BFS

# Cria grafo G
G = nx.Graph()

# Coloca os valores de karate.paj em G
G = nx.read_pajek('karate.paj')

# Chama método que retorna a BFS-tree
H = BFS(G, '1')

# Salva as profundidades cada nó de H em labels
labels = {}
for v in H.nodes():
    labels[v] = H.node[v]['depth']

# Retorna um dicionário de posições codificadas por nó
pos = nx.spring_layout(H)

# Desenha o grafo H
nx.draw(H, pos)

# Desenha com as arestas e labels
nx.draw_networkx_labels(H, pos, labels)
nx.draw_networkx_edges(H, pos)

# Exibe
plt.show()
```

Algoritmo para testar BFS para Zachary's karate club

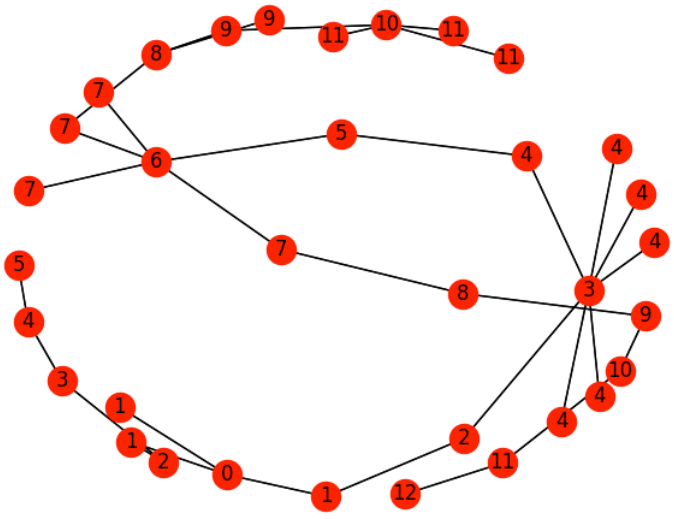


BFS-tree Zachary's karate club

testaKarateDFS.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as n
import matplotlib.pyplot as plt
from DFS import DFS                    # Importa método DFS

# Cria grafo G
G = nx.Graph()

# Coloca os valores de karate.paj em G
G = nx.read_pajek('karate.paj')

# Chama método que retorna a DFS-tree
H = DFS(G, '1')

# Salva as profundidades cada nó de H em labels
labels = {}
for v in H.nodes():
    labels[v] = H.node[v]['depth']

# Retorna um dicionário de posições codificadas por nó
pos = nx.spring_layout(H)

# Desenha o grafo H
nx.draw(H, pos)

# Desenha com as arestas e labels
nx.draw_networkx_labels(H, pos, labels)
nx.draw_networkx_edges(H, pos)

# Exibe
plt.show()
```

Algoritmo para testar DFS para Zachary's karate club



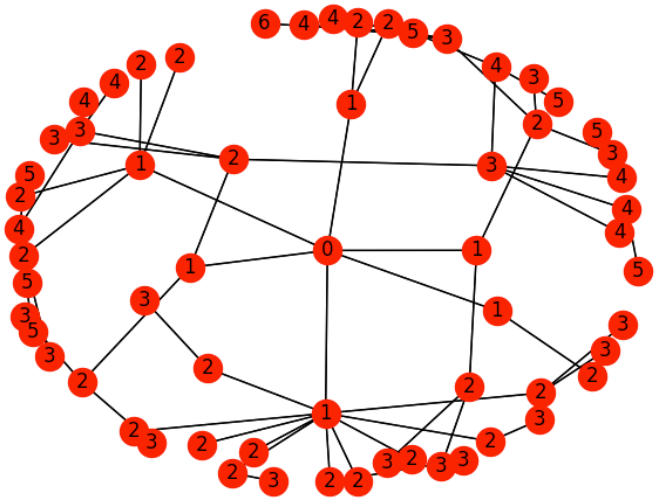
DFS-tree Zachary's karate club

Foi rodado a busca em largura e profundidade para Dolphins social network.

testaDolphinsBFS.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as n
import matplotlib.pyplot as plt
from BFS import BFS                    # Importa método BFS

# Cria grafo G
G = nx.Graph()

# Coloca os valores de dolphins.paj em G
G = nx.read_pajek('dolphins.paj')

# Chama método que retorna a BFS-tree
H = BFS(G, '0')

# Salva as profundidades cada nó de H em labels
labels = {}
for v in H.nodes():
    labels[v] = H.node[v]['depth']

# Retorna um dicionário de posições codificadas por nó
pos = nx.spring_layout(H)

# Desenha o grafo H
nx.draw(H, pos)

# Desenha com as arestas e labels
nx.draw_networkx_labels(H, pos, labels)
nx.draw_networkx_edges(H, pos)

# Exibe
plt.show()
```

Algoritmo para testar BFS para Dolphins social network

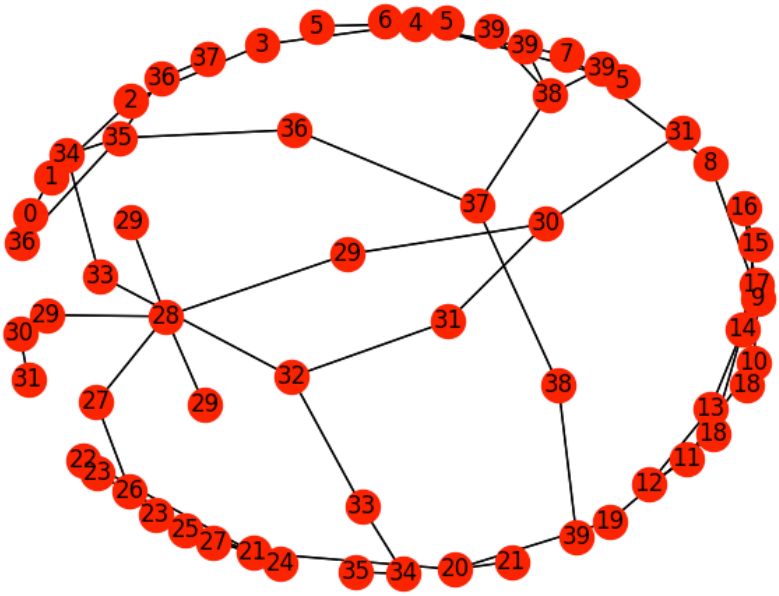


BFS-tree Dolphins social network

testaDolphinsDFS.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as n
import matplotlib.pyplot as plt
from DFS import DFS                    # Importa método DFS

# Cria grafo G
G = nx.Graph()

# Coloca os valores de dolphins.paj em G
G = nx.read_pajek('dolphins.paj')

# Chama método que retorna a DFS-tree
H = DFS(G, '0')

# Salva as profundidades cada nó de H em labels
labels = {}
for v in H.nodes():
    labels[v] = H.node[v]['depth']

# Retorna um dicionário de posições codificadas por nó
pos = nx.spring_layout(H)

# Desenha o grafo H
nx.draw(H, pos)

# Desenha com as arestas e labels
nx.draw_networkx_labels(H, pos, labels)
nx.draw_networkx_edges(H, pos)

# Exibe
plt.show()
```

Algoritmo para testar DFS para Dolphins social network



DFS-tree Dolphins social network