Graphs

May 11, 2020

[1]:
```python
import numpy as np
import random
import matplotlib.pyplot as plt
import time
import pandas as pd
```

0.1 Graph generator

Gerating graph(saturation of the graph with edges 0.6)

[2]:
```python
def Generate_file(n):
    matrix= np.random.choice([0,1],(n,n), p=[0.4, 0.6])
    matrix_symm = np.tril( matrix) + np.tril( matrix, -1).T
    with open("text.txt","w+") as f:
        np.savetxt(f,matrix_symm, fmt='%d')
```

[3]:
```python
Generate_file(7)
```

0.2 Edge list

[4]:
```python
def edge_list():
    data = np.loadtxt("text.txt")
    lst = []
    for i in range(len(data)):
        for k in range(len(data)):
            if data[i][k] == 1:
                lst.append((i,k))
    return lst
```

[5]:
```python
edgeList = edge_list()
print(edgeList)
print(len(edgeList))
```

```
[(0, 1), (0, 2), (0, 4), (0, 5), (0, 6), (1, 0), (1, 3), (2, 0), (2, 2), (2, 3),
(2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 6), (4, 0), (4, 2), (4, 3),
```

(4, 5), (4, 6), (5, 0), (5, 2), (5, 4), (5, 5), (6, 0), (6, 3), (6, 4), (6, 6)]
30

[6]:
```python
def find_edge_edgeList(edgeList,pair):
    if (pair[0],pair[1]) in edgeList or (pair[1],pair[0]) in edgeList:
        return True
    return False
```

[7]:
```python
find_edge_edgeList(edgeList,(0,1))
```

[7]: True

0.3 Adjacency list

[8]:
```python
def adjacency_list():
    data = np.loadtxt("text.txt")
    dic = {}
    for i in range(len(data)):
        temp = []
        for k in range(len(data)):
            if data[i][k] == 1:
                temp.append(k)
        dic[i] = temp
    return dic
```

[9]:
```python
adjacencyList = adjacency_list()
for i,k in adjacencyList.items():
    print(i,k)
```

```
0 [1, 2, 4, 5, 6]
1 [0, 3]
2 [0, 2, 3, 4, 5]
3 [1, 2, 3, 4, 6]
4 [0, 2, 3, 5, 6]
5 [0, 2, 4, 5]
6 [0, 3, 4, 6]
```

[10]:
```python
def find_edge_adjacencyList(adjacencyList,pair):
    return pair[0] in adjacencyList[pair[1]]
```

[11]:
```python
find_edge_adjacencyList(adjacencyList,(2,1))
```

[11]: False

0.4 Adjacency matrix

[12]:
```
def adjacency_matrix():
    data = np.loadtxt("text.txt")
    return data
```

[13]:
```
adjacencyMatrix = adjacency_matrix()
print(adjacencyMatrix)
```

```
[[0. 1. 1. 0. 1. 1. 1.]
 [1. 0. 0. 1. 0. 0. 0.]
 [1. 0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0. 1.]
 [1. 0. 1. 1. 0. 1. 1.]
 [1. 0. 1. 0. 1. 1. 0.]
 [1. 0. 0. 1. 1. 0. 1.]]
```

[14]:
```
def find_edge_adjacencyMatrix(adjacencyMatrix,pair):
    if adjacencyMatrix[pair[0]][pair[1]] == adjacencyMatrix[pair[1]][pair[0]]:
        return True
    return False
```

[15]:
```
find_edge_adjacencyMatrix(adjacencyMatrix, (2, 3))
```

[15]: True

0.5 Incidence Matrix

[16]:
```
def incidence_matrix():
    data = np.loadtxt("text.txt")
    n = len(data)
    num_edges = int(n *(n-1) /2)
    matrix = np.zeros((n,num_edges))
    col = 0
    for x in range(n):
        for y in range(x+1,n):
            if data[x][y]:
                matrix[x][col] = 1
                matrix[y][col] = 1
            col += 1
    return matrix
```

[17]:
```
incidenceMatrix = incidence_matrix()
print(incidenceMatrix)
```

```
[[1. 1. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 1. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0.]]
```

[18]:
```
def find_edge_incidenceMatrix(incidenceMatrix,pair):
    if  np.dot(incidenceMatrix[pair[0]],incidenceMatrix[pair[1]]) == 1:
        return True
    return False
```

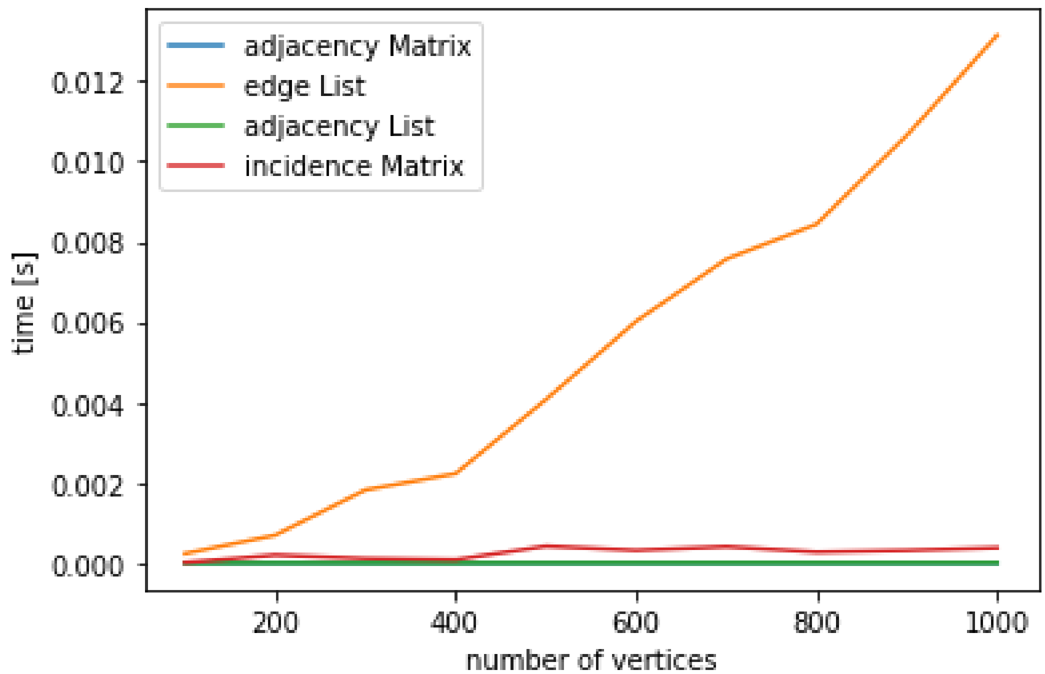
[19]:
```
find_edge_incidenceMatrix(incidenceMatrix,(4,2))
```

[19]: True

[20]:
```
def generate_numbers(n,length):
    lst = []
    for i in range(n):
        lst.append((random.randint(0,length),random.randint(0,length)))
    return lst
```

[21]:
```
def Evaluate():
    df = [[] for i in range(4)]
    functions =␣
 ↪[find_edge_adjacencyMatrix,find_edge_edgeList,find_edge_adjacencyList,find_edge_incidenceMa
    for i in range(100,1100,100):
        Generate_file(i)
        pairs = generate_numbers(i,i-1)
        representations =␣
 ↪[adjacency_matrix(),edge_list(),adjacency_list(),incidence_matrix()]
        k = 0
        for function,representation in zip(functions,representations):
            lst = []
            for pair in pairs:
                time1 = time.time()
                function(representation,pair)
                time2 = time.time() - time1
                lst.append(time2)
            df[k].append(sum(lst)/len(lst))
            k+=1
    df = list(zip(*df))
    dataframe = pd.DataFrame(df,columns = ["adjacency Matrix",\
        "edge List","adjacency List","incidence Matrix"]\
                      ,index = [i for i in range(100,1100,100)])
    plot = dataframe.plot(kind = "line")
```

```
    plot.set(xlabel="number of vertices", ylabel="time [s]")
    plt.show()
    return dataframe
Evaluate()
```



[21]:

	adjacency Matrix	edge List	adjacency List	incidence Matrix
100	1.914501e-06	0.000239	0.000002	0.000014
200	1.106262e-06	0.000696	0.000001	0.000201
300	1.180172e-06	0.001825	0.000002	0.000115
400	9.840727e-07	0.002226	0.000002	0.000088
500	1.130581e-06	0.004070	0.000003	0.000423
600	1.225471e-06	0.006016	0.000003	0.000320
700	1.010895e-06	0.007568	0.000004	0.000410
800	1.031160e-06	0.008438	0.000004	0.000277
900	1.017253e-06	0.010642	0.000004	0.000318
1000	1.111269e-06	0.013133	0.000005	0.000382

1 Conclusions

Either adjacency matrix or adjacency list outweighs both edge list and incidence matrix. Complexity for accessing the elements in the following representation is: * adjacency matrix - O(1) * edge list - O(|E|) * adjacency list - O(|V|) * incidence matrix - O(|E|)

It is important to remember that operational complexity is not the only thing that matters. Often, the key factor is the space complexity which for the aforementioned representations is as follows: * adjacency matrix - O(|V|*/V/)* edge list - O(|E|) * adjacency list - O(|V|+|E|) * incidence matrix - O(|V|*|E|)

1.1 DAG- adjacency matrix

[22]:
```
def dag_matrix(n):
    matrix = np.zeros((n,n))
    number_of_edges = int((n*(n-1)/2)*0.3)
    for i in range(number_of_edges):
        x = random.randrange(n)
        y = random.randrange(n)
        while x >= y or matrix[x][y]:
            x = random.randrange(n)
            y = random.randrange(n)
        matrix[x][y] = 1
    with open("text.txt","w+") as f:
        np.savetxt(f,matrix, fmt='%d')
```

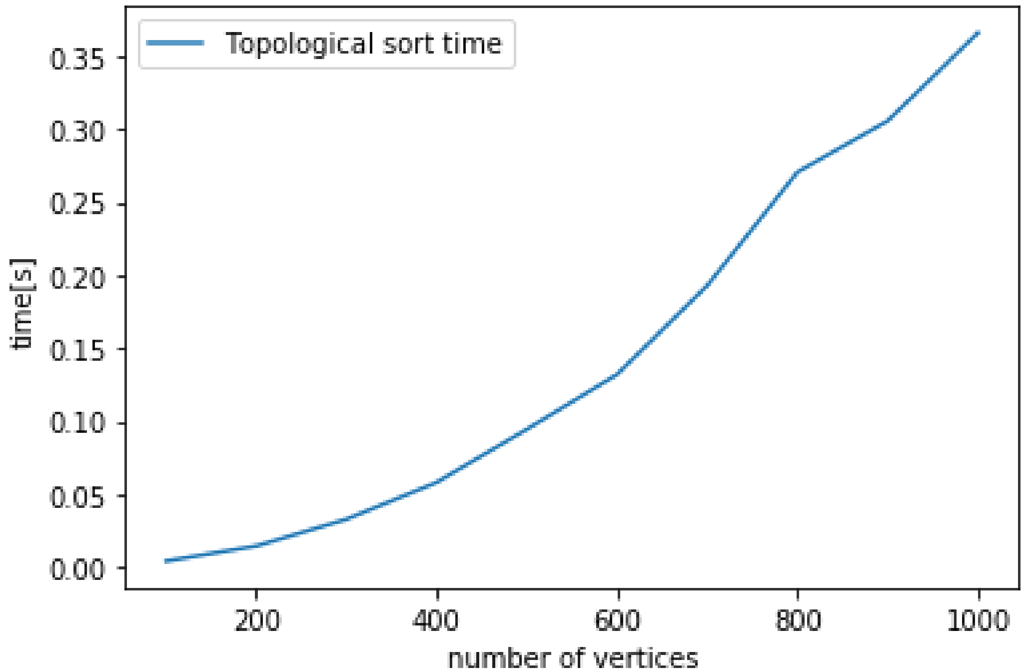
2 Topological sort for adajcency matrix

[23]:
```
def _topological_sort(i, visited, stack, graph):
    visited[i]=True
    for idx,col in enumerate(graph[i]):
        if col == 1:
            if visited[idx] == False:
                _topological_sort(idx, visited, stack, graph)
    stack.insert(0,i)

def topological_sort(graph):
    visited = [False for i in range(len(graph))]
    stack = []
    for i in range(len(graph)):
        if visited[i] == False:
            _topological_sort(i, visited, stack, graph)
    return stack
```

[24]:
```
def Evaluate2():
    df = []
    for i in range(100,1100,100):
        dag_matrix(i)
        data = np.loadtxt("text.txt")
        time1 = time.time()
        topological_sort(data)
```

```
        time2 = time.time() - time1
        df.append(time2)
    df = pd.DataFrame(df, columns = ["Topological sort time"],index = [i for i␣
 ↪in range(100,1100,100)])
    plot = df.plot(kind = "line")
    plot.set(xlabel = "number of vertices", ylabel = 'time[s]')
Evaluate2()
```



3 Conclusion

I decided to implement topological sort based on the adjacency matrix. The reason I choose this representation of the graph was simplicity to create DAG. Instead of printing the vertex immediately, program first recursively call topological sorting for all its adjacent vertices, then push it to a stack. This shows us the importance of a wise choice of graph representation. As presented in the first part, time to check the existence of the connection between two nodes strongly varies on its representation. To perform topological sort the program needs to find edges which is why the best representation for topological sort is representation with the best search performance.

Created by: Maciej Filanowicz, AI, 145462