EECE 230X – Introduction to Computation and Programming
Programming Assignment 12

- This programming assignment consists of 3 problems.
- Prerequisites: Topic 15
- Related material: Graphs: graph classes, Depth First Search (DFS), and Breadth First Search (BFS)

In this assignment, you need the the files `pa12Files.rar` available on moodle. It contains `graph.py`, `circularQueue.py` from Topic 14, and six .txt files each containing representations of six graphs.

The file `graph.py` contains:

i. An extended version of classes `DiGraph` and `UndirectedGraph` from Topic 15. Both classes are extended with a new method `draw`.

   The method `draw` produces of a visualization of the graph based on the `networkx` module (don't worry about the implementation of this function; it is just an interface with `networkx`). If `G` is a directed or undirected graph, to draw `G` using `draw`, first

```
import matplotlib.pyplot as plt
```

   Then

```
plt.figure(1) # To draw the graph on Figure 1
plt.clf()  # To clear the figure
G.draw()
plt.show() @ To show the graph
```

   If you would like to draw another graph `G2` on Figure 1, invoke `plt.clf()` to clear the figure then invoke `G2.draw()`. If you would like to draw another graph `G3` on a new or existing figure whose index is i (e.g., $i = 2$ or 3), use `plt.figure(i)` before invoking `G3.draw()`.

ii. A new function `buildGraphFromFile(fileName, undirected)`

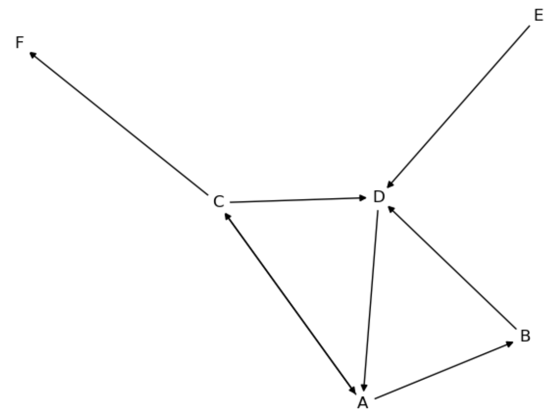
   Given a `.txt` file named `fileName`, the function `buildGraphFromFile` reads the graph form the file and returns an instance of `DiGraph` if `undirected=False`, or an instance of `UndirectedGraph` if `undirected=True`. The format of File `fileName` is as follows. Each line contains a string indicating the name of a node, followed by ":", followed by strings separated by commas indicating the names of adjacent nodes.

   Example 1 (directed graph): Assume that `DiGraph1.txt` consists of

```
A : B,C
B : D
C : D,A, F
D : A
E : D
F :
```

   Then the following code generates the below graph visualization.

```
from graph import buildGraphFromFile
import matplotlib.pyplot as plt
G = buildGraphFromFile("DiGraph1.txt", undirected = False)
plt.figure(1)
plt.clf()
G.draw()
plt.show()
```

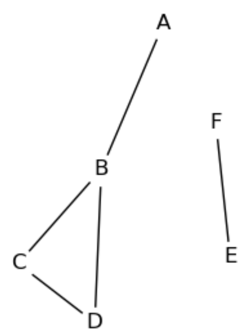


Example 2 (undirected graph): Assume that `UndirectedGraph1.txt` consists of

```
A : B
B : A,C,D
C : B,D
D : B,C
E : F
F : E
```

Then the following code generates the below graph visualization.

```
G = buildGraphFromFile("UndirectedGraph1.txt", undirected = True)
plt.figure(2)
plt.clf()
G.draw()
plt.show()
```



To make sue that the figures do not appear inside Python console, go to `Tools> Preferences> IPython console > Graphics > Graphics backend`, and set `Backend` to `Automatic`. You need to restart the kernel for this change to take effect.

iii. The function `DFSVisit` from Topic 15

Problem 1. Building graphs

a) **Build circle graph.** Write a function `buildCircleGraph(n)`, which given an integer `n`, returns the the undirected circle graph on `n` nodes $1, 2, \ldots, n$ (as integers). Assume that $n \geq 3$ and raise an assertion error if this is not the case.

Needed modules:

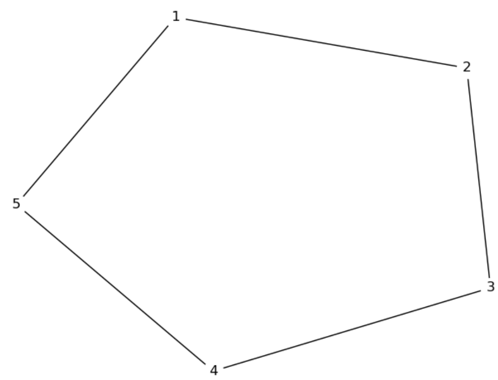
```
from graph import UndirectedGraph
import matplotlib.pyplot as plt
```

Test program:

```
G= buildCircleGraph(5)
print(G)
plt.figure(1)
plt.clf()
G.draw()
```

Output:

```
1 : 2,5
2 : 1,3
3 : 2,4
4 : 3,5
5 : 4,1
```



b) **Build graph from maze.** Recall the maze from Topic 12: a maze M is an $m \times n$ matrix with `True` and `False` values indicating whether the cells are open or closed, respectively. A maze can be viewed as an undirected graph where the nodes are the open cells and the edges connect adjacent open cells (down, right, up, and left).

Write a function `buildGraphFromMaze(M)`, which given a maze M, returns the corresponding undirected graph whose nodes are tuples (i, j) of integers.

Note: If (i, j) and (k, l) are adjacent nodes, make sure that you connect (i, j) and (k, l) only once. Otherwise, you will get the assertion error "Already connected". Namely, if you would like to use the function `adjacent(M,i,j)` from Topic 12, check if $i < k$ or $j < l$ for each (k, l) in `adjacent`(M, i, j) before connecting (i, j) and (k, l).

Needed module:

```
from graph import UndirectedGraph
```

Test program:

```
import numpy
M = [[True, False, True],
     [True, True, True],
     [True, False, True]]
     print("M:")
print(numpy.matrix(M,int))
G= buildGraphFromMaze(M)
print("G:")
print(G)
```

Output:

```
M:
[[1 0 1]
 [1 1 1]
 [1 0 1]]
G:
(0, 0) : (1, 0)
(0, 2) : (1, 2)
(1, 0) : (0, 0),(2, 0),(1, 1)
(1, 1) : (1, 0),(1, 2)
(1, 2) : (0, 2),(1, 1),(2, 2)
(2, 0) : (1, 0)
(2, 2) : (1, 2)
```

c) **Compute transpose of a directed graph.** The transpose of a directed graph G is a directed graph G^T whose nodes are the same as those of G and whose edges are reversed. Write a function `transpose(G)`, which given G, returns its transpose G^T.

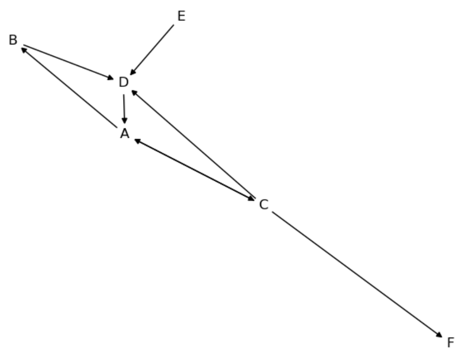
Needed modules:

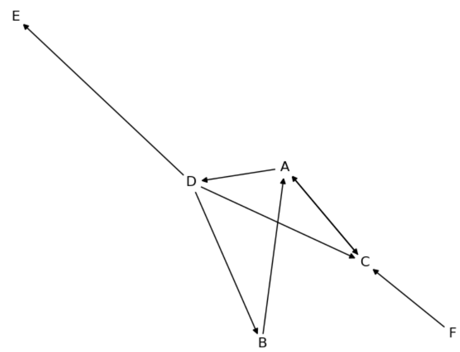
```
from graph import DiGraph, buildGraphFromFile
import matplotlib.pyplot as plt
```

Test program:

```
G = buildGraphFromFile("DiGraph1.txt", undirected = False)
# DiGraph1.txt is available in compressed folder associated with this assignment
plt.figure(1)
plt.clf()
G.draw()
GTranspose = transpose(G)
plt.figure(2)
plt.clf()
GTranspose.draw()
```

Output:





Problem 2. Extract paths

Modules needed in this poblem:

```
from graph import DFSVisit, buildGraphFromFile
import matplotlib.pyplot as plt
```

a) **Find a path from s to a given node t using DFS.** In this part, you need the fuction `DFSVisit` from Topic 15 (available in `graph.py`)

Write a function `findAPath(G,s,t)`, which takes as input arguments: a directed or undirected graph G, a node s, and a node t. If t is reachable from s, your function should return a path from s to t represented a list of nodes starting with s and ending with t. If t is not reachable from s, your function should return the empty list.

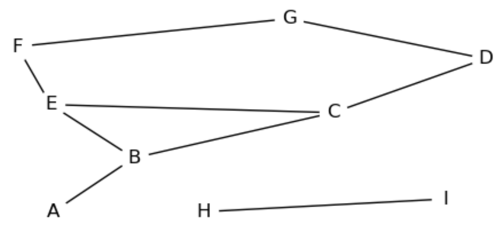
Do it as follows:

- ⋆ As in Topic 15, initialize dict `parent = {s:None}` and call the function `DFSVisit(G,s,parent)` to compute dict `parent`.
- ⋆ Write the function `extractPath(t,parent)`, which given a node t in dict `parent`, returns a path from t to s by backtracking.

Test program:

```
G = buildGraphFromFile("UndirectedGraph2.txt", undirected =True)
# UndirectedGraph2.txt is available in  compressed folder associated with this assignment
plt.figure(2)
plt.clf()
G.draw()
print(findAPath(G,'A','F'))
print(findAPath(G,'A','I'))
```

Output:




```
['A', 'B', 'C', 'D', 'G', 'F']
[]
```

b) **Find a shortest path from s to a given node t.** Repeat Part (a), but now using BFS instead of DFS to find a **shortest path**.

In particular, write a function `findShortestPath(G,s,t)`, which takes as input arguments: a directed or undirected graph G, a node s, and a node t. If t is reachable from s, your function should return a **shortest path** from s to t represented a list of nodes starting with s and ending with t. If t is not reachable from s, your function should return the empty list.

Instead of using `BFS` as a black box, you are asked to modify it by stopping the search if t is found and then using the function `extractPath(t,parent)` you wrote in Part (a).

Test program (with G as in Part (a)):

```
print(findSortestPath(G,'A','F'))
print(findSortestPath(G,'A','I'))
```

Output:

```
['A', 'B', 'E', 'F']
[]
```

Test `findShortestPath` also on the graph obtained from maze using the function `buildGraphFromMaze` in Part (b) of Problem 1.

Test program:

```
import numpy
M=[[True, False, True, True, False, True, True],
   [True, True, False, True, True, False, False],
   [False, True, True, True, True, True, True],
   [False, True, True, False, False, False, True],
   [False, True, False, True, True, False, True]]
print("M:")
print(numpy.matrix(M,int))
GMaze= buildGraphFromMaze(M)
print(findShortestPath(GMaze,(0,0),(4,6)))
```

Output:

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

Problem 3. Find connected components in an undirected graph

Let `G` be an **undirected** graph. A subset of nodes of `G` is called a connected component if it is a maximal set in which any two nodes are connected by a path.

Write a function `findConnectedComponents(G)`, which given an **undirected** graph `G`, returns a list of lists representing the connected components of `G`. Aim for $O(m+n)$ expected time.

(*Hints:*

- ⋆ Modify `DFSVisit` as follows and call it `DFSVisitModified`
- ⋆ Instead of `parent` dict, use an `index` dict initialized to the empty dict. The keys in `index` are nodes and the values are indicies of the connected components, i.e., `index[u]` should be set to the index of the compoent containing `u`.
- ⋆ Pass to `DFSVisitModified` the dict `index` as well as an int `count` initialized to zero
- ⋆ In `findConnectedComponents(G)`, loop over all nodes `u` in `G.adj`, and call `DFSVisitModified` on `u` if `u` is not in `index`)

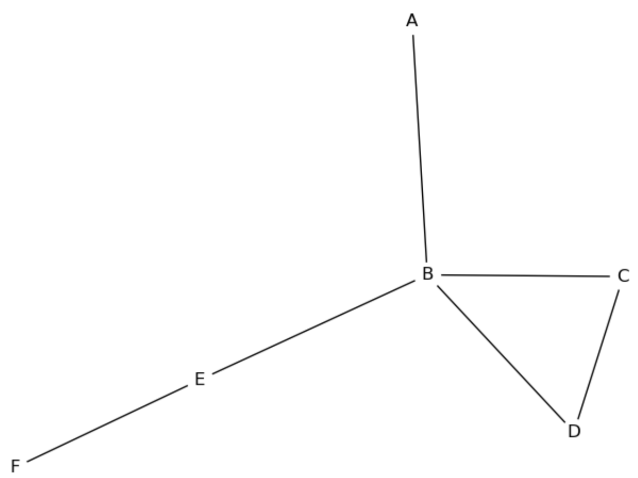
Modules needed in this poblem:

```
from graph import buildGraphFromFile
import matplotlib.pyplot as plt
```

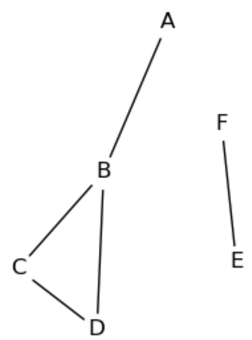
Test program:

```
G = buildGraphFromFile("UndirectedGraph3.txt", undirected =True)
plt.figure(2)
plt.clf()
G.draw()
print(findConnectedComponents(G))
G = buildGraphFromFile("UndirectedGraph1.txt", undirected =True)
plt.figure(1)
plt.clf()
G.draw()
print(findConnectedComponents(G))
G = buildGraphFromFile("UndirectedGraph4.txt", undirected =True)
plt.figure(3)
plt.clf()
G.draw()
print(findConnectedComponents(G))
```

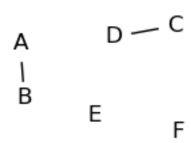
Output:



[['A', 'B', 'C', 'D', 'E', 'F']]



[['A', 'B', 'C', 'D'], ['E', 'F']]



[['A', 'B'], ['C', 'D'], ['E'], ['F']]