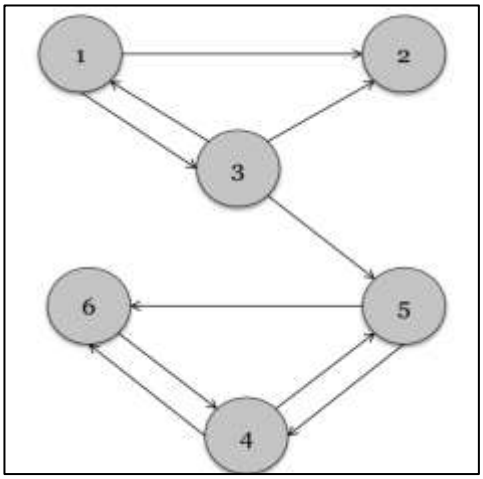
Implement page rank algorithm

- **Q1. Estimate the page rank for the given directed graph representing web of six pages and damping factor is 0.9 and Max. Iterations are 2.**



First find adjacency matrix (A sample code for adjacency matrix, not for the above graph)

You need to initialize your graph as directed before you start adding edges to it:

```
gd = Graph(directed=True)
gd.add_vertices(5)
gd.add_edges([(0,1),(1,2)])
print(gd.get_adjacency())
# [[0, 1, 0, 0, 0]
#  [0, 0, 1, 0, 0]
#  [0, 0, 0, 0, 0]
#  [0, 0, 0, 0, 0]
#  [0, 0, 0, 0, 0]]
```

The `.to_directed()` method that Matthew mentioned won't do what you want, because it creates edges in both directions for each undirected edge:

```
print(g.is_directed())    # your original undirected example graph
# False

g.to_directed()

print(g.is_directed())
# True

print(g.get_adjacency())  # adjacency matrix is still symmetric
# [[0, 1, 0, 0, 0]
#  [1, 0, 1, 0, 0]
#  [0, 1, 0, 0, 0]
#  [0, 0, 0, 0, 0]
#  [0, 0, 0, 0, 0]]
```

You can prevent both directed edges from being created by passing `.to_directed(mutual=False)`, but in this case igraph just picks an arbitrary single direction for each edge. The fundamental problem is that information about the directions of edges is simply not stored if the graph is undirected.

Find stochastic matrix (Delete those pages not having out-going links and get stochastic matrix)

Adjacency Matrix A for the above Graph:

	P1	P2	P3	P4	P5	P6
P1	0	1/2	1/2	0	0	0
P2	0	0	0	0	0	0
P3	1/3	1/3	0	0	1/3	0
P4	0	0	0	0	1/2	1/2
P5	0	0	0	1/2	0	1/2
P6	0	0	0	1	0	0

Stochastic Matrix

	P1	P2	P3	P4	P5	P6
P1	0	1/2	1/2	0	0	0
P2	1/6	1/6	1/6	1/6	1/6	1/6
P3	1/3	1/3	0	0	1/3	0
P4	0	0	0	0	1/2	1/2
P5	0	0	0	1/2	0	1/2
P6	0	0	0	1	0	0

Find transpose

A^T

0	1/6	1/3	0	0	0
1/2	1/6	1/3	0	0	0
1/2	1/6	0	0	0	0
0	1/6	0	0	1/2	1
0	1/6	0	1/2	0	0

0	1/6	0	1/2	1/2	0

numpy.matrix.transpose

`matrix.transpose`(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters:	**axes** *: None, tuple of ints, or n ints* None or no argument: reverses the order of the axes. tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis. *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)
Returns:	**out** *: ndarray* View of *a*, with axes suitably permuted.

See also

`ndarray.T`

Array property returning the array transposed.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
```

```
        [2, 4]])
```

Then find the following calculations for a fixed K value

(1-d)*E/n + d*A^T where d=0.9

(1-0.9)* $\frac{1}{6}$ +0.9 * A^T

1/60 +

0	0.15	0.3	0	0	0
0.45	0.15	0.3	0	0	0
0.45	0.15	0	0	0	0
0	0.15	0	0	0.45	0.9
0	0.15	0.3	0.45	0	0
0	0.15	0	0.45	0.45	0

P=

0.016	0.016	0.316	0.016	0.016	0.016
0.46	0.166	0.316	0.016	0.016	0.016
0.46	0.166	0.016	0.016	0.016	0.016
0.016	0.166	0.016	0.016	0.46	0.916
0.016	0.166	0.316	0.46	0.016	0.016
0.016	0.166	0.016	0.46	0.46	0.016

Iteration 1:

P*

1
1
1
1
1
1
1

0.016	0.016	0.316	0.016	0.016	0.016		1		0.862
0.46	0.166	0.316	0.016	0.016	0.016		1		0.99
0.46	0.166	0.016	0.016	0.016	0.016		1		0.69
0.016	0.166	0.016	0.016	0.46	0.916	*	1	=	1.44
0.016	0.166	0.316	0.46	0.016	0.016		1		0.84
0.016	0.166	0.016	0.46	0.46	0.016		1		0.92

Iteration 2:

P1	0.016	0.016	0.316	0.016	0.016	0.016		0.862		0.340
P2	0.46	0.166	0.316	0.016	0.016	0.016		0.99		0.831
P3	0.46	0.166	0.016	0.016	0.016	0.016		0.69		0.475
P4	0.016	0.166	0.016	0.016	0.46	0.916	*	1.44	=	0.970
P5	0.016	0.166	0.316	0.46	0.016	0.016		0.84		0.830
P6	0.016	0.166	0.016	0.46	0.46	0.016		0.92		0.880

After 2 iterations:

We would then continue this iterating until the values are approximately stable, and we would be able to determine the importance ranking using the resulting vector. With this, we can see that even with a small count of 2 iterations, our vector was already converging towards the eigenvector. Since this is the importance vector of our network, we can see that the PageRank importance ranking of our pages would be

P4 > P6 > P2 > P5 > P3 > P1