In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

1a)

In [3]:
```python
# Circle topology
# Unweighted adjacency matrix

# Option 1: Manually enter the entries
Atilde = np.array(
          [[0,1,0,0,0,0,0,1],
           [1,0,1,0,0,0,0,0],
           [1,1,0,1,1,0,0,0],
           [0,0,1,0,1,0,0,0],
           [0,0,0,1,0,1,0,0],
           [0,0,0,0,1,0,1,0],
           [0,0,0,0,0,1,0,1],
           [1,0,0,0,0,0,1,0]])

# Option 2: or you can exploit the patterns
# Atilde = np.zeros((8,8))
# for i in range(8): #
#     Atilde[i,(i+1)%8] = 1
#     Atilde[i,(i-1)%8] = 1
# Atilde[2,0] = 1
# Atilde[2,4] = 1

print('Unweighted adjacency matrix')
print(Atilde)
print(' ')
```

```
Unweighted adjacency matrix
[[0 1 0 0 0 0 0 1]
 [1 0 1 0 0 0 0 0]
 [1 1 0 1 1 0 0 0]
 [0 0 1 0 1 0 0 0]
 [0 0 0 1 0 1 0 0]
 [0 0 0 0 1 0 1 0]
 [0 0 0 0 0 1 0 1]
 [1 0 0 0 0 0 1 0]]
```

1b)

In [6]:
```python
# Find weighted adjacency matrix
# option 1: normalize columns with a for loop
A = np.zeros((8,8), dtype=float)
for k in range(8):
    norm = np.sum(Atilde[:,k])
    if norm != 0:
        A[:,k] = np.round(Atilde[:,k] / norm,2)
    else:
        A[:,k] = Atilde[:,k]
```

```python
    # option 2: normalize using numpy.sum() and broadcasting, in a single line
# A = ???

print('Weighted adjacency matrix')
print(A)
```

```
Weighted adjacency matrix
[[0.   0.5  0.   0.   0.   0.   0.   0.5 ]
 [0.33 0.   0.5  0.   0.   0.   0.   0.  ]
 [0.33 0.5  0.   0.5  0.33 0.   0.   0.  ]
 [0.   0.   0.5  0.   0.33 0.   0.   0.  ]
 [0.   0.   0.   0.5  0.   0.5  0.   0.  ]
 [0.   0.   0.   0.   0.33 0.   0.5  0.  ]
 [0.   0.   0.   0.   0.   0.5  0.   0.5 ]
 [0.33 0.   0.   0.   0.   0.   0.5  0.  ]]
```

In [ ]:

```python

```

1c) and 1d)

In [15]:

```python
# Power method

b0 = 0.125*np.ones((8,1))
print('b0 = ', b0)
print(' ')

b1 = np.dot(A, b0)
print('b1 = ', b1)
print(' ')

b = b0.copy()
for k in range(1000):
    b = np.dot(A, b)

print('1000 iterations')
print('b = ',b)
```

```
b0 =  [[0.125]
 [0.125]
 [0.125]
 [0.125]
 [0.125]
 [0.125]
 [0.125]
 [0.125]]

b1 =  [[0.125  ]
 [0.10375]
 [0.2075 ]
 [0.10375]
 [0.125  ]
 [0.10375]
 [0.125  ]
 [0.10375]]

1000 iterations
b =  [[0.00112892]
 [0.00150232]
 [0.00225262]
 [0.00150232]
 [0.00112892]
 [0.00075029]
 [0.00075203]
 [0.00075029]]
```

1e) Explanation goes here.

In [ ]:
```
# Nodes 3, 2, 4 (in that order) are the most important.
```

2a)

In [16]:
```
# Hub topology

Atildehub = np.array(
         [[0,0,0,0,0,0,0,0,1],
          [1,0,0,0,0,0,0,0,1],
          [0,0,0,0,0,0,0,0,1],
          [0,0,0,0,0,0,0,0,1],
          [0,0,0,0,0,0,0,0,1],
          [0,0,0,0,0,0,0,0,1],
          [0,0,0,0,0,0,0,0,1],
          [0,0,0,0,0,0,0,0,1],
          [1,1,1,1,1,1,1,1,0]])

print('Unweighted adjacency matrix')
print(Atildehub)
print(' ')
```

```
Unweighted adjacency matrix
[[0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1]
 [1 1 1 1 1 1 1 1 0]]
```

2b)

In [17]:
```python
# find weighted adjacency matrix

Ahub = np.zeros((9,9), dtype=float)
for k in range(9):
    norm = np.sum(Atildehub[:,k])
    Ahub[:,k] = Atildehub[:,k] / norm

print('Weighted adjacency matrix')
print(Ahub)
```

```
Weighted adjacency matrix
[[0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.5   0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.125]
 [0.5   1.    1.    1.    1.    1.    1.    1.    0.   ]]
```

2c) and 2d)

In [18]:
```python
b0 = (1/9)*np.ones((9,1))
print('b0 = ', b0)
print(' ')

bhub1 = Ahub @ b0
print('bhub1 = ', bhub1)
print(' ')

bhub = b0.copy()
for k in range(1000):
    bhub = Ahub @ bhub

print('1000 iterations')
print('bhub = ', bhub)
print(' ')

bhubr = b0.copy()
for k in range(100):
```

```
    bhubr = Ahub @ bhubr

print('100 iterations')
print('bhubr = ',bhubr)
print(' ')

bhub3 = b0.copy()
for k in range(101):
    bhub3 = Ahub @ bhub3

print('101 iterations')
print('bhub3 = ',bhub3)
```

```
b0 =  [[0.11111111]
 [0.11111111]
 [0.11111111]
 [0.11111111]
 [0.11111111]
 [0.11111111]
 [0.11111111]
 [0.11111111]
 [0.11111111]]

bhub1 =  [[0.01388889]
 [0.06944444]
 [0.01388889]
 [0.01388889]
 [0.01388889]
 [0.01388889]
 [0.01388889]
 [0.01388889]
 [0.83333333]]

1000 iterations
bhub =  [[0.06060606]
 [0.09090909]
 [0.06060606]
 [0.06060606]
 [0.06060606]
 [0.06060606]
 [0.06060606]
 [0.06060606]
 [0.48484848]]

100 iterations
bhubr =  [[0.06065482]
 [0.09093172]
 [0.06065482]
 [0.06065482]
 [0.06065482]
 [0.06065482]
 [0.06065482]
 [0.06065482]
 [0.48448454]]

101 iterations
bhub3 =  [[0.06056057]
 [0.09088798]
 [0.06056057]
 [0.06056057]
 [0.06056057]
 [0.06056057]
 [0.06056057]
 [0.06056057]
 [0.48518805]]
```

Complete 2e and 2f below.

In [19]:
```
# 2e comment:
# Node 9 is clearly the most important as every other node points to it.
# Node 2 is slightly more important than the rest of the branch nodes because node

# 2f comment:
# After 1000 iterations score9 is 0.4848 which rounds to 0.485 (I assume this to be
# After 101 iterations, score9 is 0.4852 which rounds to 0.485.
# So 101 iterations is required for 3 decimal point accuracy.
```

In [21]:
```
# Problem 3 comment:
# The sign of the singular vectors is not unique.
# If you consider the case where u_i = -u_i and v_i = -v_i,
# then sigma_i * -u_i * -v_i^T = sigma_i * u_i * v_i^T
# Which is the valid singular value decomposition for X.
```

In [ ]:
```
```