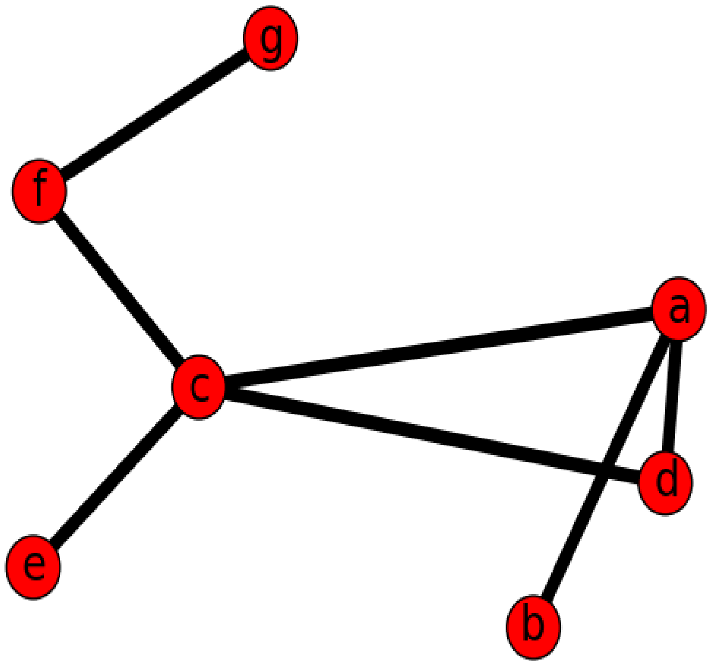
CMSC5733 Social Computing

Tutorial IV: HW2 Solution and More about NetworkX

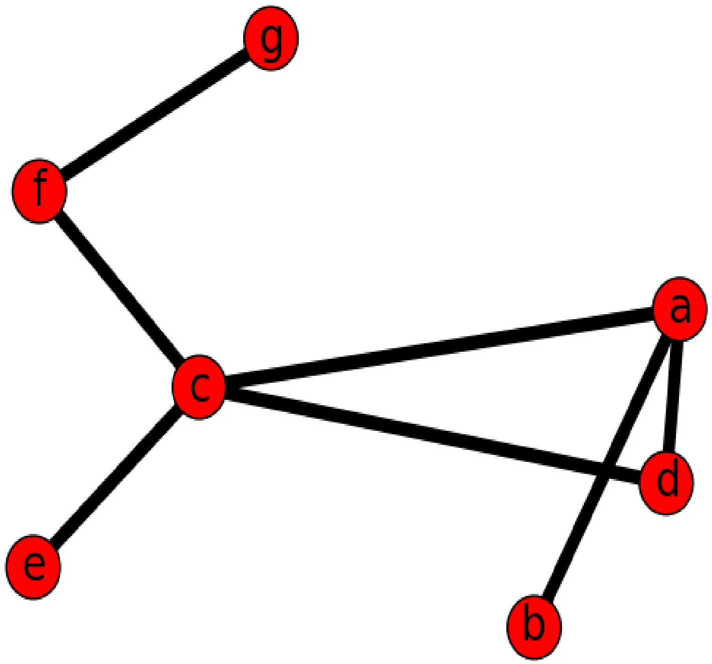
Shenglin Zhao
The Chinese University of Hong Kong
slzhao@cse.cuhk.edu.hk

1.1 Radius



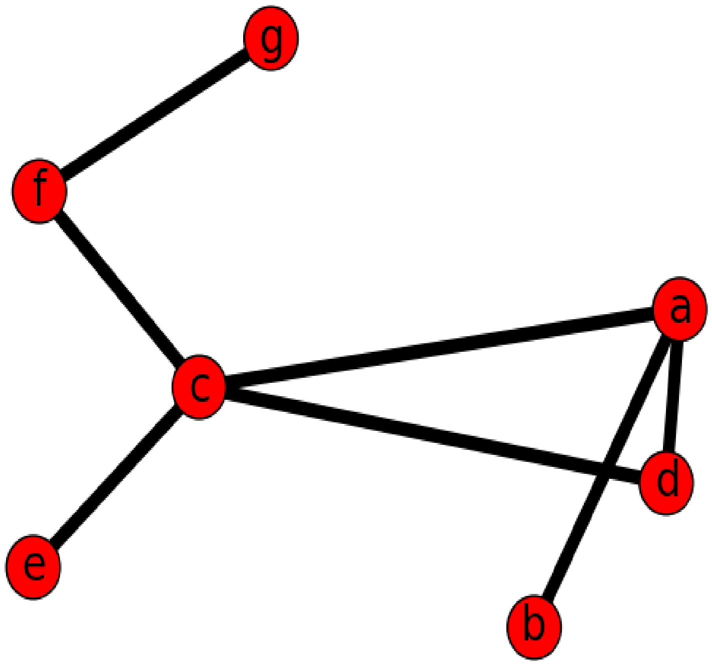
- Radius of one node means the largest direct path length from the node to other nodes.
- Answer:
  - a: 3 (a-c-f-g)
  - d: 3 (d-c-f-g)
  - f: 3 (f-d-a-d)

1.2 Diameter



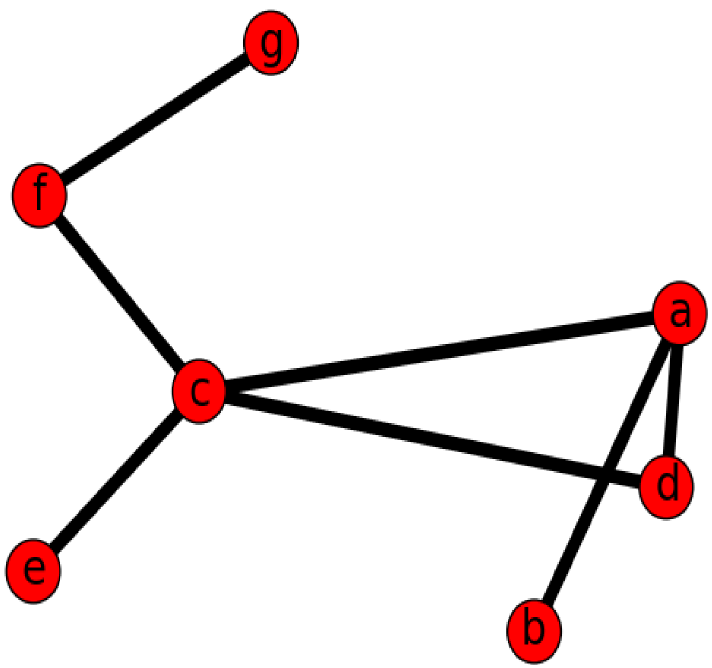
- Diameter means the largest one among all direct path length of any node pair, namely, longest shortest path.
- Answer: 4
  - b-a-c-f-g

1.3 Center



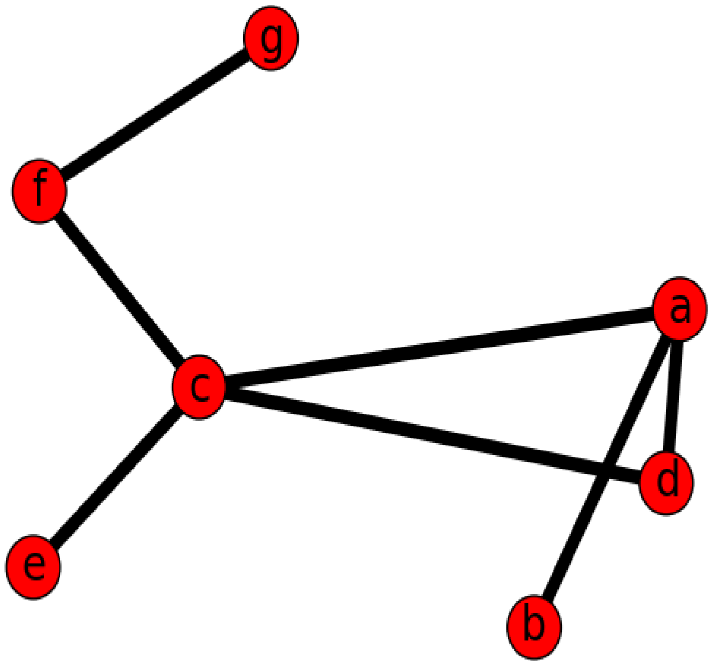
- Center is the node with smallest radius.
- Answer: c
  - a: 3
  - b:4
  - c:2
  - d:3
  - e:3
  - f:3
  - g:4

1.4 Center



- C is the best place.
- Because node c is the center of the graph. It can reach other nodes in smallest average numbers of link, and that will reduce transit time most.

1.5 Adjacency Matrix



Answer:

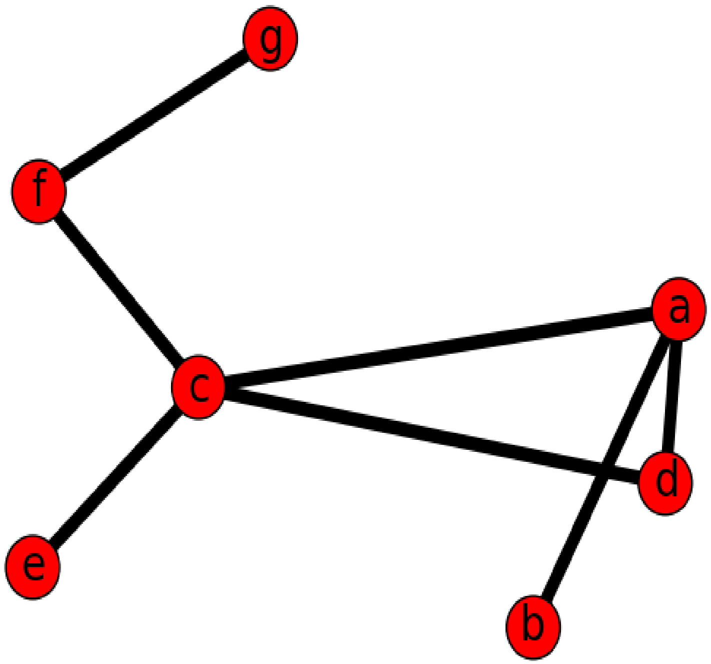
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

1.5 Laplacian Matrix

The *Laplacian matrix* of graph G, namely, $L(G)$, is a combination of the connection matrix and (diagonal) degree matrix: $L = C - D$, where D is a diagonal matrix and C is the connection (adjacency) matrix

$$d_{i,j} = \begin{cases} \text{degree}(v_i) & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

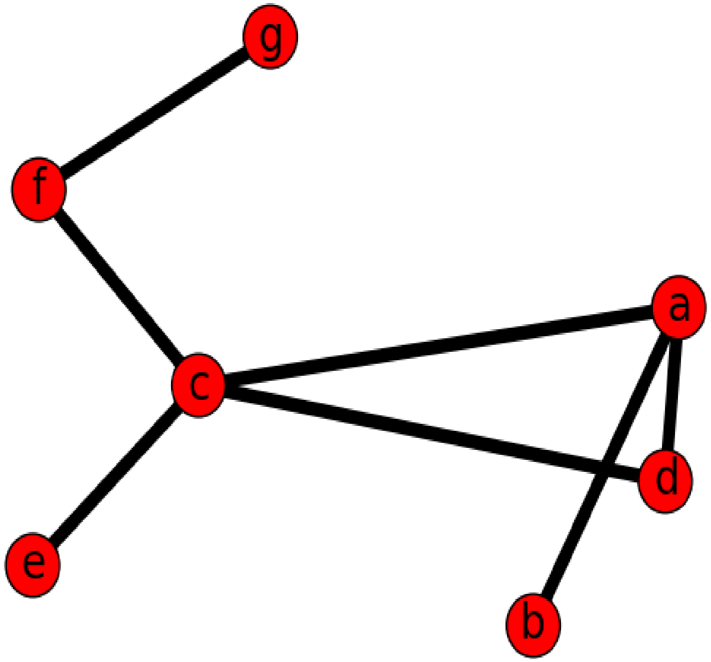
1.5 Laplacian Matrix



- D=

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

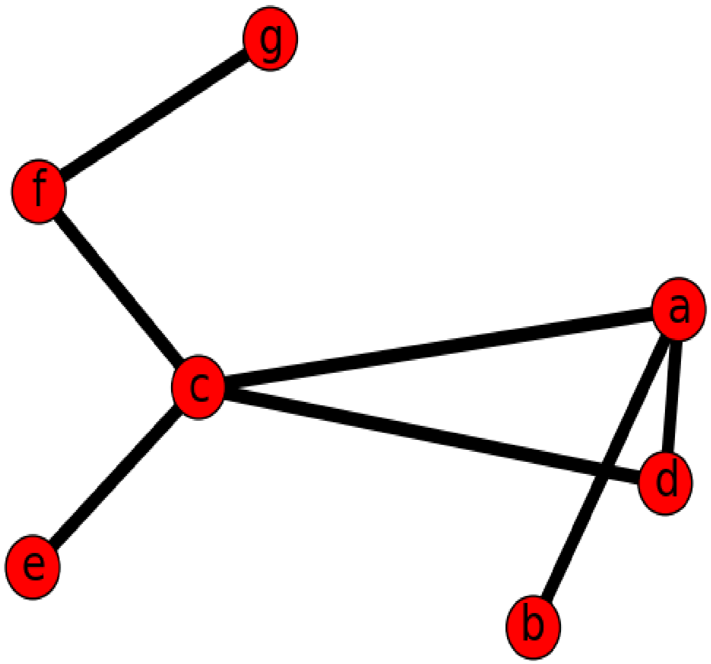
1.5 Laplacian Matrix



- Answer: C-D=

$$\begin{bmatrix} -3 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -4 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

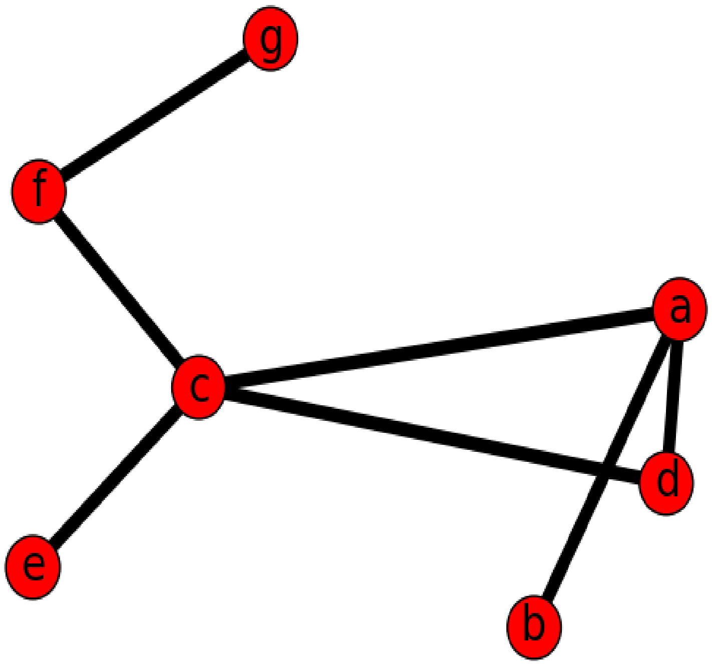
1.5 Laplacian Matrix



- Answer:

$$\begin{bmatrix} -3 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -4 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

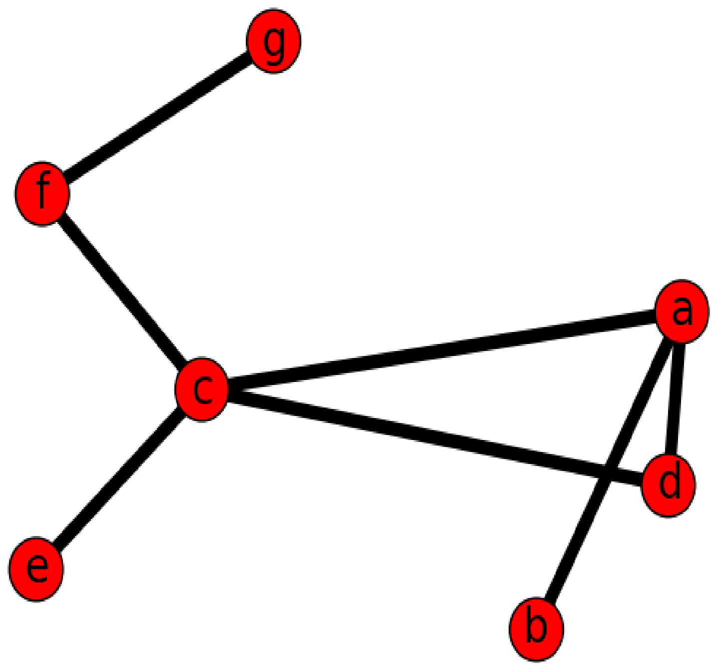
1.7



- The deletion of which vertex will make the network unconnected?
- Answer: (c is the center)
  - f : (g vs. a,b,c,d,e)

NetworkX for Q1

- Draw the graph



Drawing in NetworkX

`draw` (G[, pos, ax, hold])	Draw the graph G with Matplotlib.
`draw_networkx` (G[, pos, arrows, with_labels])	Draw the graph G using Matplotlib.
`draw_networkx_nodes` (G, pos[, nodelist, ...])	Draw the nodes of the graph G.
`draw_networkx_edges` (G, pos[, edgelist, ...])	Draw the edges of the graph G.
`draw_networkx_labels` (G, pos[, labels, ...])	Draw node labels on the graph G.
`draw_networkx_edge_labels` (G, pos[, ...])	Draw edge labels.
`draw_circular` (G, **kwargs)	Draw the graph G with a circular lay
`draw_random` (G, **kwargs)	Draw the graph G with a random lay
`draw_spectral` (G, **kwargs)	Draw the graph G with a spectral lay
`draw_spring` (G, **kwargs)	Draw the graph G with a spring layou
`draw_shell` (G, **kwargs)	Draw networkx graph with shell layo
`draw_graphviz` (G[, prog])	Draw networkx graph with graphviz

Radius for nodes in Networkx

eccentricity

eccentricity(*G, v=None, sp=None*) [source]

Return the eccentricity of nodes in G.

The eccentricity of a node v is the maximum distance from v to all other nodes in G.

Parameters:

- **G** (*NetworkX graph*) – A graph
- **v** (*node, optional*) – Return value of specified node
- **sp** (*dict of dicts, optional*) – All pairs shortest path lengths as a dictionary of dictionaries

Returns: **ecc** – A dictionary of eccentricity values keyed by node.

Return type: dictionary

```
In [6]: nx.eccentricity(G)
Out[6]: {'a': 3, 'b': 4, 'c': 2, 'd': 3, 'e': 3, 'f': 3, 'g': 4}
```

Radius of Graph

radius

radius(*G, e=None*) [source]

Return the radius of the graph G.

The radius is the minimum eccentricity.

Parameters:
- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns: **r** – Radius of graph

Return type: integer

```
In [4]: nx.radius(G)
Out[4]: 2
```

Diameter

diameter

diameter(*G, e=None*) [source]

Return the diameter of the graph G.

The diameter is the maximum eccentricity.

Parameters:
- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns: **d** – Diameter of graph

Return type: integer

```
In [7]: nx.diameter(G)
Out[7]: 4
```

Center

center

center(*G, e=None*) [source]

Return the center of the graph G.

The center is the set of nodes with eccentricity equal to radius.

Parameters:
- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns: **c** – List of nodes in center

Return type: list

```
In [8]: nx.center(G)
Out[8]: ['c']
```

Adjacency Matrix

adjacency_matrix

`adjacency_matrix`(*G, nodelist=None, weight='weight'*) [source]

Return adjacency matrix of G.

Parameters:

G : graph

A NetworkX graph

nodelist : list, optional

The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

weight : string or None, optional (default='weight')

The edge data key used to provide each value in the matrix. If None, then each edge has weight 1.

Returns:

A : SciPy sparse matrix

Adjacency matrix representation of G.

Adjacency Matrix

```
In [9]: A = nx.adjacency_matrix(G, ['a','b', 'c', 'd','e','f','g'])

In [10]: A.todense()
Out[10]:
matrix([[0, 1, 1, 1, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0],
        [1, 0, 0, 1, 1, 1, 0],
        [1, 0, 1, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 1, 0]])
```

```
In [11]: print A
  (0, 1)        1
  (0, 2)        1
  (0, 3)        1
  (1, 0)        1
  (2, 0)        1
  (2, 3)        1
  (2, 4)        1
  (2, 5)        1
  (3, 0)        1
  (3, 2)        1
  (4, 2)        1
  (5, 2)        1
  (5, 6)        1
  (6, 5)        1
```

```
In [12]: A = nx.adjacency_matrix(G)

In [13]: A.todense()
Out[13]:
matrix([[0, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 1, 1, 0, 1],
        [1, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0],
        [1, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 1, 0, 0, 0, 1, 0]])

In [14]: G.nodes()
Out[14]: ['a', 'c', 'b', 'e', 'd', 'g', 'f']
```

Laplacian Matrix

laplacian_matrix

`laplacian_matrix`(*G, nodelist=None, weight='weight'*) [source]

Return the Laplacian matrix of G.

The graph Laplacian is the matrix L = D - A, where A is the adjacency matrix and D is the diagonal matrix of node degrees.

Parameters:

G : graph

A NetworkX graph

nodelist : list, optional

The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

weight : string or None, optional (default='weight')

The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns:

L : SciPy sparse matrix

The Laplacian matrix of G.

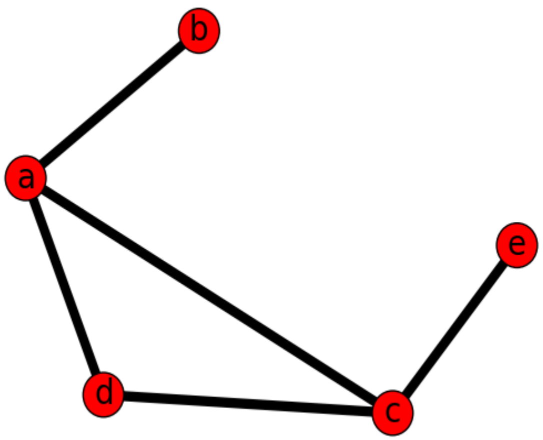
Laplacian Matrix

```
In [17]: L = nx.laplacian_matrix(G, ['a','b','c','d','e','f','g'])

In [18]: L.todense()
Out[18]:
matrix([[ 3, -1, -1, -1,  0,  0,  0],
        [-1,  1,  0,  0,  0,  0,  0],
        [-1,  0,  4, -1, -1, -1,  0],
        [-1,  0, -1,  2,  0,  0,  0],
        [ 0,  0, -1,  0,  1,  0,  0],
        [ 0,  0, -1,  0,  0,  2, -1],
        [ 0,  0,  0,  0,  0, -1,  1]])

In [19]: -L.todense()
Out[19]:
matrix([[-3,  1,  1,  1,  0,  0,  0],
        [ 1, -1,  0,  0,  0,  0,  0],
        [ 1,  0, -4,  1,  1,  1,  0],
        [ 1,  0,  1, -2,  0,  0,  0],
        [ 0,  0,  1,  0, -1,  0,  0],
        [ 0,  0,  1,  0,  0, -2,  1],
        [ 0,  0,  0,  0,  0,  1, -1]])
```

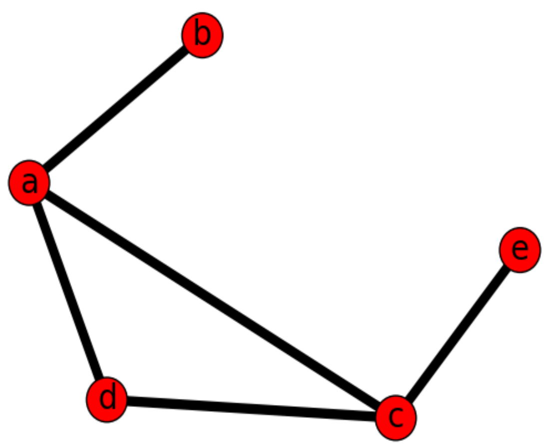
2.1 Density



$$\text{Density} = \frac{2|E|}{|V|(|V|-1)}$$

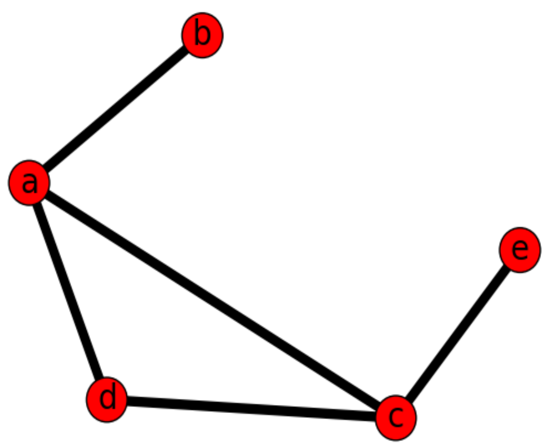
- Answer:
  - 2 * 5/(5*4) = 0.5

2.2 Degree Sequence



- Degree sequence: g=[d1, d2,…,dn] defines a degree sequence containing the degree values of all n nodes in G.
  - [3, 1, 3, 2, 1], from a to e

2.3 Average Path Length



- The average path length of the graph is the average of all shorted paths

$$l_G = \frac{1}{n \cdot (n-1)} \cdot \sum_{i,j} d(v_i, v_j)$$

AB = 1, AD = 1, AC = 1,AE = 2;

BC = 2, BD = 2, BE = 3;

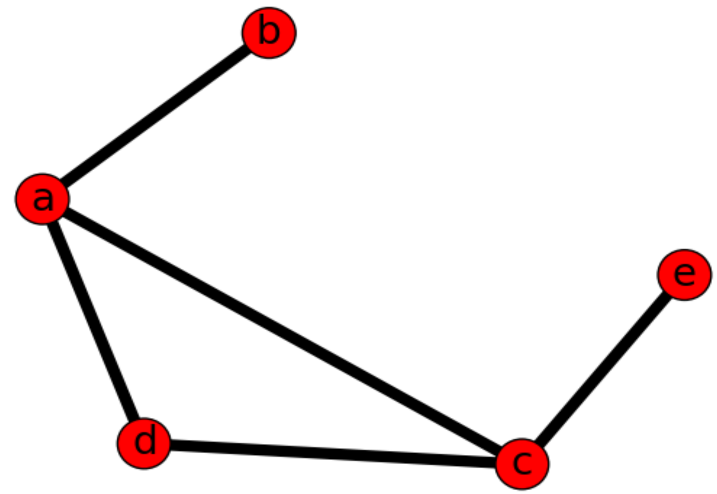
CD = 1, CE = 1;

DE = 2;

– Answer: 16*2/(5*4)=1.6

NetworkX for Q2

- Draw the graph



Density

density

`density(G)` [source]

Return the density of a graph.

The density for undirected graphs is

$$d = \frac{2m}{n(n-1)},$$

and for directed graphs is

$$d = \frac{m}{n(n-1)},$$

where n is the number of nodes and m is the number of edges in G.

Notes

The density is 0 for a graph without edges and 1 for a complete graph. The density of multigraphs can be higher than 1.

Self loops are counted in the total number of edges so graphs with self loops can have density higher than 1.

```
In [21]: nx.density(G)
Out[21]: 0.5
```

Average Path Length

average_shortest_path_length

`average_shortest_path_length(G, weight=None)` [source]

Return the average shortest path length.

The average shortest path length is

$$a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}$$

where V is the set of nodes in G, $d(s,t)$ is the shortest path from s to t, and n is the number of nodes in G.

Parameters:

G : NetworkX graph

weight : None or string, optional (default = None)

If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Raises:

NetworkXError:

if the graph is not connected.

```
In [23]: nx.average_shortest_path_length(G)
Out[23]: 1.6
```

3 Cluster Coefficient

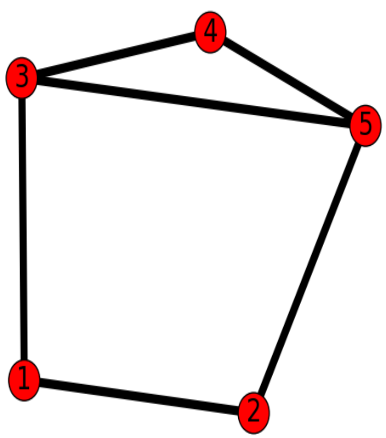
- For a node u, suppose that the neighbors share c links, then the cluster coefficient of node u,

$$Cc(u) = \frac{2c}{\text{degree}(u)(\text{degree}(u) - 1)}$$

- The cluster coefficient of the graph is average cluster coefficient over all nodes,

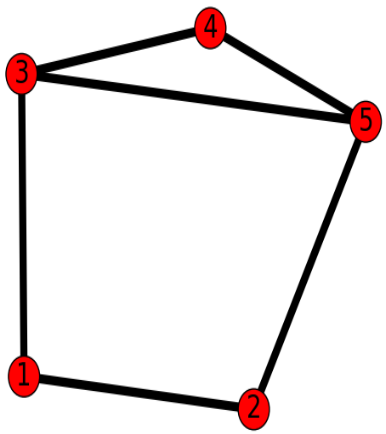
$$CC(G) = \sum_{i=1}^{n} \frac{Cc(v_i)}{n}$$

3.1 Cluster Coefficient



- Answer:
  - cc(1) = 0
  - cc(2) = 0
  - cc(3) = 2*1/(3*2) = 1/3
    - neighbors: 1, 4, 5
    - shared links: 1, (4,5)
    - degree(3): 3
  - cc(4) = 2*1/(2*1) = 1
    - neighbors: 4, 5
    - shared links: 1, (4,5)
    - degree(4): 2
  - cc(5) = 1/3
    - symmetric with node 3

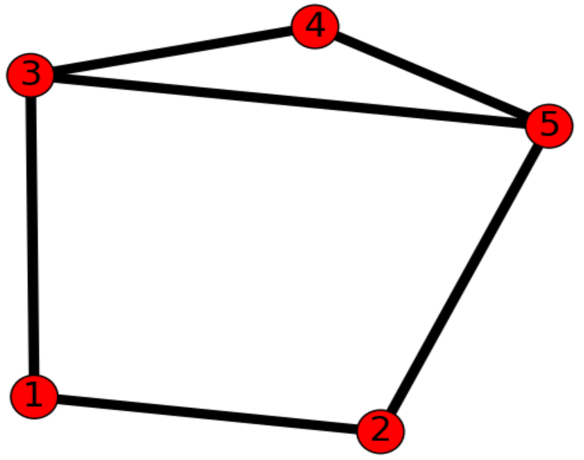
3.2 Cluster Coefficient



- CC(G)
- Answer:
  - (0+0+1/3+1+1/3)/5=1/3

NetworkX for Q3

- Draw the graph



Coefficient

clustering

`clustering`(*G, nodes=None, weight=None*) [source]

Compute the clustering coefficient for nodes.

For unweighted graphs, the clustering of a node u is the fraction of possible triangles through that node that exist,

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)},$$

where $T(u)$ is the number of triangles through node u and $deg(u)$ is the degree of u.

For weighted graphs, the clustering is defined as the geometric average of the subgraph edge weights [R202]

```
In [27]: nx.clustering(G)
Out[27]:
{'1': 0.0,
 '2': 0.0,
 '3': 0.3333333333333333,
 '4': 1.0,
 '5': 0.3333333333333333}
```

The edg

$\hat{w}_{uv} =$

The value of c_u is assigned to 0 if $deg(u) < 2$.

Graph Coefficient

average_clustering

`average_clustering`(*G, nodes=None, weight=None, count_zeros=True*) [source]

Compute the average clustering coefficient for the graph G.

The clustering coefficient for the graph is the average,

$$C = \frac{1}{n}\sum_{v \in G} c_v,$$

where n is the number of nodes in G.

Parameters:

G : graph

nodes : container of nodes, optional (default=all nodes in G)

Compute average clustering for nodes in this container.

weight : string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

count_zeros : bool (default=False)

If False include only the nodes with nonzero clustering in the average.

Returns:

avg : float

Average clustering

```
In [25]: nx.average_clustering(G)
Out[25]: 0.3333333333333333
```

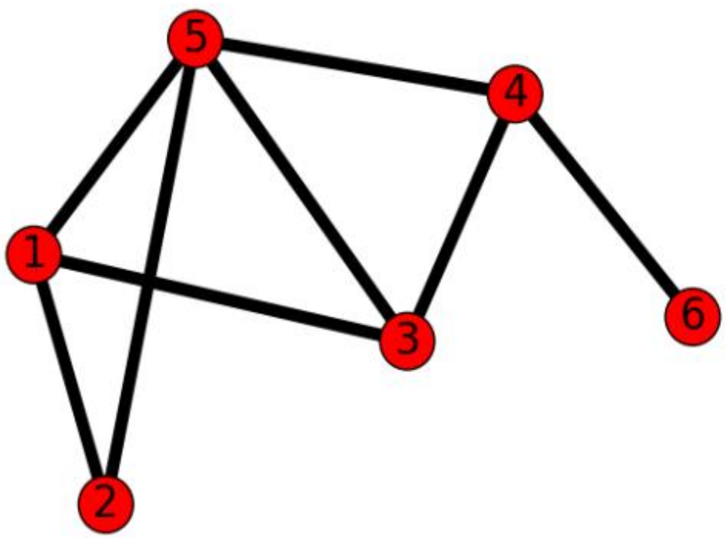
4.1 Closeness

- Closeness of a node u is the reciprocal of sum of the shortest path distance from u to all n-1 other nodes.

$$C(u) = \frac{1}{\sum_{v=1}^{n} d(v,u)}$$

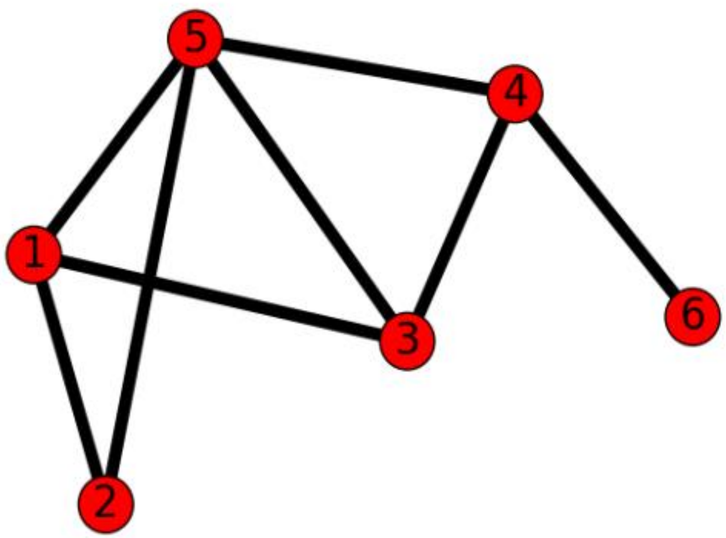
d(v, u) is the shortest path distance between v and u, and n is the number of nodes in the graph.

4.1 Closeness



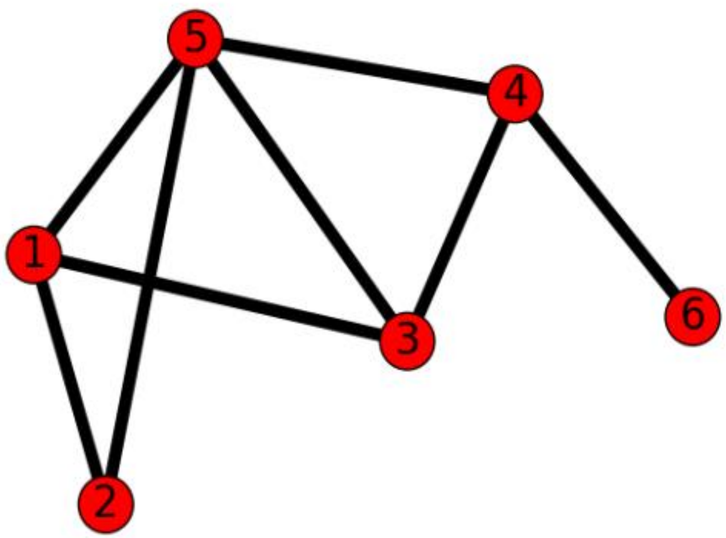
- Answer:
- shortest paths from node 3:
  - 3-1:1,
  - 3-2:2,
  - 3-4:1,
  - 3-5:1,
  - 3-6:2,
  - sum of shortest paths: 1+2+1+1+2= 7
  - closeness = 1/7

4.1 Closeness



- Answer:
- shortest paths from node 5:
  - 5-1:1,
  - 5-2:1,
  - 5-3:1,
  - 5-4:1,
  - 5-6:2,
  - sum of shortest paths: 1+1+1+1+2= 6
  - closeness = 1/6

4.1 Normalized closeness



- Closeness is normalized by the sum of minimum possible distance n-1

$$C(u) = \frac{n-1}{\sum_{v=1}^{n} d(v,u)}$$

- Answer:
  - C(3) = (6-1)/7 = 5/7 = 0.714
  - C(5) = (6-1)/6 = 5/6 = 0.833

4.2 Betweenness

Betweenness Centrality of a node counts the number of times that a node lies along the shortest path between two others vertices in the graph. It is defined as

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{g\sigma_{st}}.$$

where σ_{st} is the number of shortest paths from s to t and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through a vertex v.

4.2 Betweenness

- 1. For each pair of vertices (s, t), compute the shortest paths between them.
- 2. For each pair of vertices (s, t), determine the fraction of shortest paths that pass through the vertex in question (here, vertex v).
- 3. Sum this fraction over all pairs of vertices (s, t).

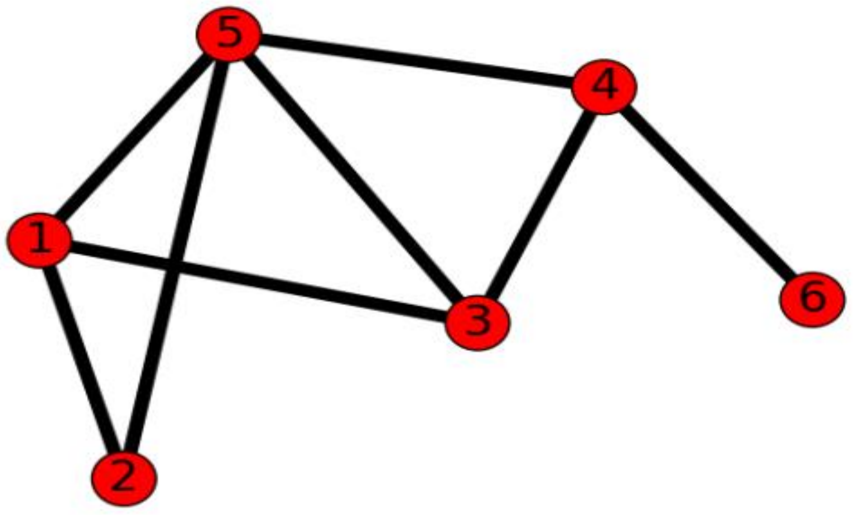
4.2 Betweenness

- For node 3,

Pairs	Shortest paths	Total	Via 3	Fraction
(1,4)	(1,5,4), (1,3,4)	2	1	0.5
(1,6)	(1,5,4,6), (1,3,4,6)	2	1	0.5

betweenness =

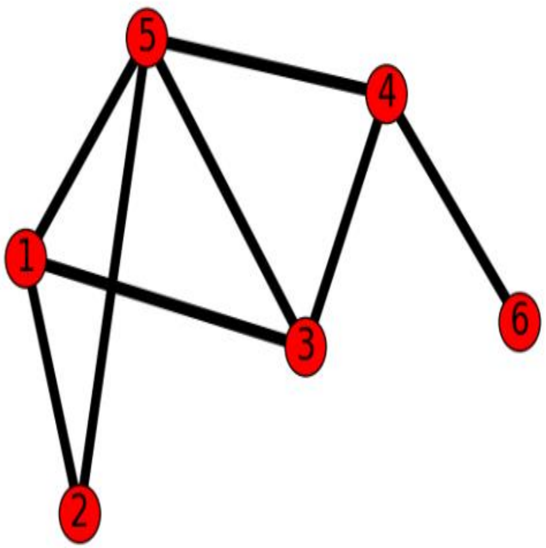
0.5+0.5 = 1



Betweenness

- For node 5,

Pairs	Shortest paths	Total	Via 5	Fraction
(1,4)	(1,5,4), (1,3,4)	2	1	0.5
(1,6)	(1,5,4,6), (1,3,4,6)	2	1	0.5
(2,3)	(2,1,3), (2,5,3)	2	1	0.5
(2,4)	(2,5,4)	1	1	1
(2,6)	(2,5,4,6)	1	1	1



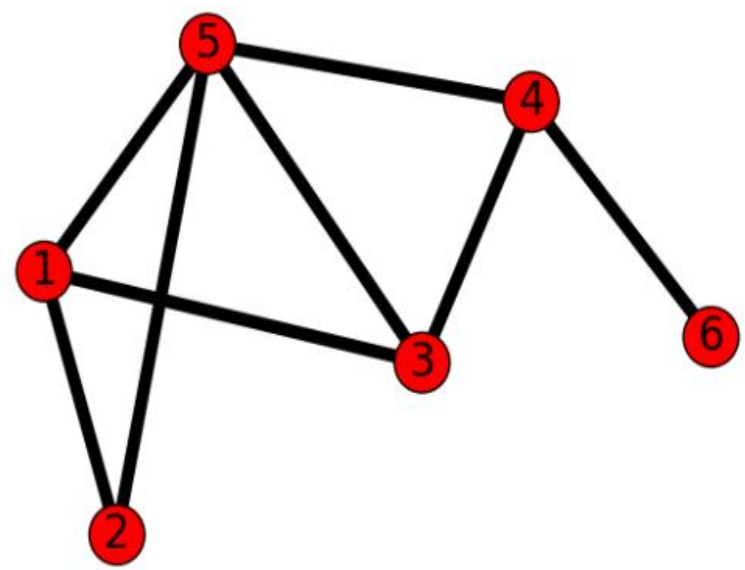
- betweenness = 0.5+0.5+0.5+1+1=3.5

Normalized Betweenness

- Betweenness is normalized by 2/((n-1)(n-2)) for undirected graphs, and 1/((n-1)(n-2)) for directed graphs
- Betweenness(3) = 2*1/((6-1)*(6-2)) = 0.1
- Betweenness(5) = 3.5*2/(5*4) = 0.35

NetworkX for Q4

- Draw the graph



closeness

closeness_centrality

`closeness_centrality`(*G, u=None, distance=None, normalized=True*) [source]

Compute closeness centrality for nodes.

Closeness centrality [R174] of a node u is the reciprocal of the sum of the shortest path distances from u to all $n-1$ other nodes. Since the sum of distances depends on the number of nodes in the graph, closeness is normalized by the sum of minimum possible distances $n-1$.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)},$$

where $d(v,u)$ is the shortest-path distance between v and u, and n is the number of nodes in the graph.

Notice that higher values of closeness indicate higher centrality.

Parameters:

G : graph

A NetworkX graph

u : node, optional

Return only the value for node u

distance : edge attribute key, optional (default=None)

Use the specified edge attribute as the edge distance in shortest path calculations

normalized : bool, optional

If True (default) normalize by the number of nodes in the connected part of the graph.

Returns:

nodes : dictionary

Dictionary of nodes with closeness centrality as the value.

Closeness

```
In [39]: nx.closeness_centrality(G, u='3')
Out[39]: 0.7142857142857143

In [40]: nx.closeness_centrality(G, u='5')
Out[40]: 0.8333333333333334
```

```
In [41]: nx.closeness_centrality(G)
Out[41]:
{'1': 0.625,
 '2': 0.5555555555555556,
 '3': 0.7142857142857143,
 '4': 0.7142857142857143,
 '5': 0.8333333333333334,
 '6': 0.45454545454545453}
```

Betweenness

betweenness_centrality

`betweenness_centrality`(*G, k=None, normalized=True, weight=None, endpoints=False, seed=None*) [source]

Compute the shortest-path betweenness centrality for nodes.

Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

where V is the set of nodes, $\sigma(s, t)$ is the number of shortest (s, t)-paths, and $\sigma(s, t|v)$ is the number of those paths passing through some node v other than s, t. If $s = t$, $\sigma(s, t) = 1$, and if $v \in s, t$, $\sigma(s, t|v) = 0$ [R172].

Betweenness

Parameters: **G** : graph

A NetworkX graph

k : int, optional (default=None)

If k is not None use k node samples to estimate betweenness. The value of k <= n where n is the number of nodes in the graph. Higher values give better approximation.

normalized : bool, optional

If True the betweenness values are normalized by $2/((n-1)(n-2))$ for graphs, and $1/((n-1)(n-2))$ for directed graphs where n is the number of nodes in G.

weight : None or string, optional

If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.

endpoints : bool, optional

If True include the endpoints in the shortest path counts.

Returns: **nodes** : dictionary

Dictionary of nodes with betweenness centrality as the value.

```
In [43]: nx.betweenness_centrality(G)
Out[43]: {'1': 0.05, '2': 0.0, '3': 0.1, '4': 0.4, '5': 0.35000000000000003, '6'
: 0.0}
```

Q5

- Answer: toroidal network