Lab 2

Tanner Kogel tjk190000

MECH 6317.001: Dynaics of Complex Networks & Systems

imoport all important libraries

In [1]:
```
import networkx as nx
import sys
import numpy
import itertools
import asyncio
```

Section 6.9: Degree

Identify an example type of network of when it is a good thing to be a high degree node and an example of when it is a bad thing to be a high degree node in a network.

An example of a network when it is a good thing to be a high degree node would be a netowrk where the nodes are airports and the edges are direct flights, because traveling would become very easy

An example of a network when it is a bad thing to be a high degree would be a network where the nodes are people and the edges are transmissions of diseases

Describe or draw an example of a network in which a particular node has very few connections, but it could be argued that it is a very important node. Justify your reasoning.

An example of a network in which a particular node has very few connections, but it could be argued that is a very important node is the previous example network, where nodes are people and the edges are trasmissions of diseases, because a node with a small degree (few neighbors) could be an important node by possibly having an immunity and be able to help find a cure.

Explain what equation (6.25) in Newman actually means - i.e., what is a simpler way to explain what the two summations mean?

The incoming degree of a node in a directed network is the sum of the values in the row that represents that node, and the outgoing degree of that node is the sum of the values in the column that represents that node.

Section 6.10-6.11: Paths & Components

write down the adjacency matrix of the network shown in the diagram of the lab manual

In [2]:
```
A = [[0,0,0,0,0],
     [1,0,0,0,0],
     [1,1,0,0,0],
     [0,1,1,0,0],
     [0,0,1,1,0]]
print(A)
```

```
[[0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 1, 0, 0, 0], [0, 1, 1, 0, 0], [0, 0, 1, 1, 0]]
```

write down the adjacency matrix twice and multiply them to get A^2

In [3]:
```
print(A)
print(A)
print(numpy.dot(A,A))
```

```
[[0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 1, 0, 0, 0], [0, 1, 1, 0, 0], [0, 0, 1, 1, 0]]
[[0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 1, 0, 0, 0], [0, 1, 1, 0, 0], [0, 0, 1, 1, 0]]
[[0 0 0 0 0]
 [0 0 0 0 0]
 [1 0 0 0 0]
 [2 1 0 0 0]
 [1 2 1 0 0]]
```

each nonzero element describes the existent of n paths from node j to node i of length 2

the paths described in each cell of A^2 are described here:

0,2

0 --> 1 --> 2

0,3

0 --> 1 --> 3

0 --> 2 --> 3

0,4

0 --> 2 --> 4

1,3

1 --> 2 --> 3

1,4

1 --> 2 --> 4

1 --> 3 --> 4

2,4

2 --> 3 --> 4

use the vector x = [1,0,0,0,0].T to analyze A

multiply Ax, A^2x=A(Ax), and A^3x=A(A(Ax))

In [4]:
```
x = [[1],
     [0],
     [0],
     [0],
     [0]]
```

```
A2 = numpy.dot(A,A)
A3 = numpy.dot(A2,A)
print(numpy.dot(A,x))
print(numpy.dot(A2,x))
print(numpy.dot(A3,x))
```

```
[[0]
 [1]
 [1]
 [0]
 [0]]
[[0]
 [0]
 [1]
 [2]
 [1]]
[[0]
 [0]
 [0]
 [1]
 [3]]
```

The multiplication by x, returns the first column of the matrix it is being multiplied by. In terms of the network meaning, it represents the number of r-length paths from node 0 to each node in the graph where r is the power of the adjacency matrix A. For example, in the column vector printed second, the first two values are zero, meaning there are no paths of length two between node 0 and nodes 0 and 1. The values of one in indeces 2 and 4 indicate one path of length two between node 0 and nodes 2 and 4, and finally the value of two in index 3 indicates two paths of length two between node 0 and node 3.

repeat this process with a the new definition of x = [0,0,1,0,0].T

In [5]:
```
x = [[0],
     [0],
     [1],
     [0],
     [0]]
A2 = numpy.dot(A,A)
A3 = numpy.dot(A2,A)
print(numpy.dot(A,x))
print(numpy.dot(A2,x))
print(numpy.dot(A3,x))
```

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

This observation does generalize, because this new definition of x takes out the third column of the matrix it is multiplied by, meaning that the meanings of these values are the same as seen in the previous section, but for node 2 instead of node 0.

Filled in Stub Code

In [6]:
```
import networkx as nx
import sys
sys.path.append('../d3networkx/')
import d3networkx as d3nx
from d3graph import D3Graph, D3DiGraph
from numpy import *
from time import time
import asyncio

def square_grid(n,d3,G,x0=100,y0=100,w=50):
    if G is None:
        G = D3Graph()
    # find the dimensions for the grid that are as close as possible
    num_rows = int(floor(sqrt(n)))
    while n % num_rows != 0:
        num_rows += 1
    num_cols = int(n/num_rows)

    # Add all the nodes
    G.add_nodes_from(range(n))

    # Add the edges and position the nodes
```

```
    for i in range(num_rows):
        for j in range(num_cols):
            n = num_cols*i + j
            d3.position_node(n,x0+i*w,y0+j*w)
            if i < num_rows-1:
                G.add_edge(n,n+num_cols) # add edge down
            if j < num_cols-1:
                G.add_edge(n,n+1) # add edge right

async def propagate(G,d3,x,steps,slp=0.5,keep_highlights=False,update_at_end=False):
    interactive = d3.interactive
    d3.set_interactive(False)
    A = nx.to_numpy_array(G).T  # adjacency matrix
    d3.highlight_nodes_by_index(list(where(x>0)[0]))
    d3.update()
    await asyncio.sleep(slp)
    cum_highlighted = sign(x)
    for i in range(steps): # the brains
        x = sign(dot(A,x)) # the brains
        cum_highlighted = sign(cum_highlighted+x)
        if not update_at_end:
            if not keep_highlights:
                d3.clear_highlights()
            d3.highlight_nodes_by_index(list(where(x>0)[0]))
            d3.update()
            await asyncio.sleep(slp)
    if update_at_end:
        if not keep_highlights:
            d3.clear_highlights()
            d3.highlight_nodes_by_index(list(where(x>0)[0]))
        else:
            d3.highlight_nodes_by_index(list(where(cum_highlighted>0)[0]))
        d3.update()
    d3.set_interactive(interactive)
    if keep_highlights:
        return cum_highlighted
    else:
        return x
```

This next line starts up the visualizer. It will start some background code that sends data to the visualizer and then it will open a new browser window where the visualizer will live. Once you have the visualizer running, you can leave it running for the entire session, so don't re-run this block. If you close the `visualizer.html` (or hit refresh), you will need to reestablish this connection. In this case,

you should click the refresh button in the Jupyter notebook (not for the webpage) to restart the kernel (which will clear your variables and Python environment).

In [7]:
```
d3 = await d3nx.create_d3nx_visualizer(canvas_size=(1000,800))
#d3 = await d3nx.create_d3nx_visualizer(canvas_size=(1200,1000))
d3.clear()
```

```
websocket server started...networkx connected...visualizer connected...
```

Launching the Visualizer Manually

If the visualizer does not launch automatically, then you'll need to open it manually. After running the line above, use the following line to determine the communication port that the visualizer is using:

The port is a 4-digit number. Go to the file *visualizer.html* in the d3networkx folder. Double click on it to open it (do not open it in JupyterLabs). In the url, add the following (without the quotes) to the end of the url: "&port=1234" and replace 1234 with the 4-digit port above. A different port is selected each time you run the `d3 = await d3nx.create_d3nx_visualizer()` command. So as long as you don't run that command again (or restart the kernel, or close JupyterLab, that port should still continue to work)

Grid Network

In [8]:
```
d3.clear()
d3.set_interactive(False)
G = D3Graph()
d3.set_graph(G)
square_grid(144,d3,G,x0=75,y0=70)
d3.update()
```

The above code block creates a square graph of which 100 nodes in the center of the graph have a degree of 4, 40 nodes along the edges of the graph have a degree of 3, and 4 nodes at the corners of the graph have a degree 2. A histogram of these findings are in the supplimental material following the python code for this lab.

In [9]:
```
x = zeros((G.number_of_nodes(),1))
x[0] = 1
await propagate(G,d3,x,10,slp=1);
```

the above code block creates a pattern where at each step it highlights the nodes that can be reached in r steps from node 0. the pattern stops where it does because it is set to start at r=0, and end at r=10, as indicated in the call of the function. The propogation

could go further if the number 10 was changed to a higher number, the function would continue looping for longer paths

Directed Network

In [10]:
```
d3.clear()
G = D3DiGraph(nx.read_weighted_edgelist('lab2.edgelist',create_using=nx.DiGraph))
d3.set_graph(G)
d3.update()
```

add code to propogate this network starting at node 0 for 10 steps

In [11]:
```
x = zeros((G.number_of_nodes(),1)) # create a column vector x of size n populated by zeros
x[0] = 1                           # set the first value of x to 1
await propagate(G,d3,x,10,slp=1);  # propagate the network from node 0 for 10 steps
```

the end result of the above code block is that the pattern is shown for the first four steps and no nodes are highlighted for the following steps

A^10 is a 4x4 square matrix populated entirely by zeros

This happens from a network path point of view because this directed network has no directed loops, so any directed path eventually gets to node 4, and once the directed path reaches node 4 it cannot move to another node by following the directions of the edges. The longest path from node 0 to node 4 has a length of 4, so after the 4-step iteration of the loop, no nodes are highlighted.

this happens from a linear algebra point of view because A is a matrix that is (or could be in a general case) organized in a way such that all nonzero elements are above the diagonal of the matrix, making A a noninvertable, or singular matrix

propogate the network starting at node 1 wih keep_highlights=True to show the out-component of node 1

In [12]:
```
d3.clear_highlights()                                                          # clear highlights from visualizer
x = zeros((G.number_of_nodes(),1))                                             # create a column vector of size n  popul
x[1] = 1                                                                       # set the index 1 value of x to 1
x = await propagate(G,d3,x,10,slp=1,keep_highlights=True,update_at_end=True); # propagate the network from node 1 for 1
print(x)                                                                       # print column vector of all highlighted
```

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

The out-component is what I predicted. Due to all numbers pointing to the two nodes of greater value than them I hypothesized that the out-component would include everything all nodes of index 1 or greater

Without changing the code within the propagate function, how could we use it to find in-components instead of out-components

create a new graph G_reverse that is identical to G, but all the edges point in the opposite direction

In [13]:
```
d3.clear_highlights()               # clear highlights from visualizer
G_reverse = d3nx.DiGraph()          # initialize a graph G_reverse to point in opposite direction as G
G_reverse.add_nodes_from(G.nodes()) # ensure nodes are the same
for i in G.edges():                 # loop for every edge in G
    G_reverse.add_edge(i[1],i[0])   # add an edge that points the opposite direction

x = zeros((G_reverse.number_of_nodes(),1))                                    # create a zero  column vector of
x[1] = 1                                                                      # set the index 1 value of x to 1
x = await propagate(G_reverse,d3,x,10,slp=1,keep_highlights=True,update_at_end=True); # propagate the network from node
print(x)                                                                      # print column vector of all high
```

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

there is no strongly connected component in this network because the graph is acyclic

E. coli Protein Network

In [23]:
```
d3.clear()
G = D3DiGraph(nx.read_weighted_edgelist('ecoli.edgelist',create_using=nx.DiGraph))
d3.set_interactive(False)
d3.set_graph(G)
d3.set_interactive(True)
d3.update()
print('Ecoli has %i nodes.' % G.number_of_nodes())
```

```
Ecoli has 418 nodes.
```

find the out-components of node with index 2 and print out the size of the component

In [24]:
```
d3.clear_highlights()                                                    # clear highlihghts from the visualizer
x = zeros((G.number_of_nodes(),1))                                       # create a column vector of size n  populated
x[2] = 1                                                                 # set the index 2 value of x to 1
x = await propagate(G,d3,x,417,keep_highlights=True,update_at_end=True); # propagate the network from node 2 for n ste
print(size(list(where(x>0)[0])))                                         # print size of the component
```

```
3
```

find the out-components of node with index 16 and print out the size of the component

In [25]:
```
d3.clear_highlights()                                                    # clear highlihghts from the visualizer
x = zeros((G.number_of_nodes(),1))                                       # create a column vector of size n  populated
x[16] = 1                                                                # set the index 2 value of x to 1
x = await propagate(G,d3,x,417,keep_highlights=True,update_at_end=True); # propagate the network from node 2 for n ste
print(size(list(where(x>0)[0])))                                         # print size of the component
```

```
22
```

the minimum value of steps that guarantees that you will find the entire out-component is n-1, where n is the number of nodes in the network

create a function to find the diameter and path of a graph with multiple componenets

In [26]:
```
def diameter2(G):
    spaths = dict(nx.all_pairs_shortest_path(G)) # get shortest path between all pairs
    diameter = 0                                 # set diameter value at 0
    for i in spaths.values():                    # loop for each key in the outer dict & give the value
        for j in i.values():                     # loop for each key in the inner dict & give the value
            if size(j) > diameter:               # path of note (longer than previous longest path)
                diameter = size(j)               # record length of longest shortest path (so far)
                path = j                         # record longest shortest path (so far)
    d3.highlight_nodes(path)                     # highlight path on visualizer
    d3.update()                                  # load new highlights to visualizer
    return diameter, path                        # return appropriate values for diameter and path

d3.clear_highlights()                      # clear visualizer of previous highlights
[diameter,path] = diameter2(G)             # use diameter2 function to find diameter and path
print('Diameter of graph: %i' % diameter) # print diameter of E. Coli graph
print('Path of Diameter:')                 # print path diameter label
print(path)                                # print path of diameter
```

```
Diameter of graph: 5
Path of Diameter:
['190', '292', '136', '137', '133']
```

Section 6.12: Flows & Cut Sets

create functions that create the best and worst running time for the augmented path algorithm for a graph of n nodes

the worst_graph function creates a graph where every node points to every other node in the network, thus having the maximum number of edges (without self-edges) and maximum number of independent paths

the best_graph function creates a graph where every node i points to i+1 within the range n, the simplest way to make a (weakly) connected graph with only one independent path between any two nodes

In [18]:
```python
# create directed graph that will have worst computing time for augmented path algorithm
def worst_graph(n):
    G = nx.DiGraph()                        # initialize G as a directed graph
    G.add_nodes_from(range(n))              # add all nodes in range to G
    for i in G.nodes():                     # loop for every node in G (predecessor)
        for j in G.nodes():                 # loop for every node in G (successor)
            if i != j:                      # prevent self-loop
                G.add_edge(i,j,capacity=1) # add directed edge from every node to every node that is not itself
    return G

# create directed graph that will have best computing time for augmented path algorithm
def best_graph(n):
    G = nx.DiGraph()                       # initialize G as a directed graph
    G.add_nodes_from(range(n))             # add all nodes in range to G
    for i in G.nodes():                    # loop for every node in G (predecessor)
        if i > 0:                          # do not create path for last node
            G.add_edge(i-1,i,capacity=1) # add directed from node i to node i+1 in range
    return G

# worst graph check
G = worst_graph(1000)                                                  # create worst graph with 1000 nodes
start_time = time()                                                    # save start time
worst_cut_value, worst_partition = nx.minimum_cut(G,0,999)             # run the augmented path algorithm with wor
print('min cut for worst graph took %1.2f seconds' % (time() - start_time)) # print run time of algorithm for worst gra

# best graph check
```

```
G = best_graph(1000)                                                    # create worst graph with 1000 nodes
start_time = time()                                                     # save start time
best_cut_value, best_partition = nx.minimum_cut(G,0,999)                # run the augmented path algorithm with wors
print('min cut for best graph took %1.2f seconds' % (time() - start_time)) # print run time of algorithm for worst grap
```

```
min cut for worst graph took 6.57 seconds
min cut for best graph took 0.02 seconds
```

explain why a project activity network should be an acyclic directed network and why cycles would be a bad way to organize it.

a project activity network should be an acyclic directed network because the graph should be sequential, move from start to finish without backtracking at all, because that could cause a loop, preventing the project from finishing at all

use the Bellman-Ford shortest path algorithm to find the length and activity sequence of the critical path in the pert.gml activity network

In [70]:
```
G = nx.read_gml('pert.gml','name') # read in the network shown in the lab manual

critical_path = nx.bellman_ford_path(G,'Lead time','Leave site')          # find critical path of activity sequence
critical_length = nx.bellman_ford_path_length(G,'Lead time','Leave site') # find length of critical path
# print out length and activity sequence of critical path
print('The critical path of the activity sequence:', critical_path)
print ('The length of the activity sequence is %i units' % critical_length)
```

```
The critical path of the activity sequence: ['Lead time', 'Obtain valves', 'Fit valves', 'Finish valve chambers', 'Leav
e site']
The length of the activity sequence is 48 units
```

calculate the minimum cut set and the cut set size from “Lead time” to “Leave site”

In [71]:
```
cut_value, partition = nx.minimum_cut(G, 'Lead time', 'Leave site',capacity = 'weight') # use minimum_cut to find minim
reachable, non_reachable = partition                                                    # divide partition to two funct

# use functino from help page to find output cutset_trial
cutset_trial = set()
for u, nbrs in ((n, G[n]) for n in reachable):
    cutset_trial.update((u, v) for v in nbrs if v in non_reachable)

# upon studying the output of cutset_trial, it seems that the first entry of each element of
# sorted(cutset_trial) makes up the minium cut set of the network
```

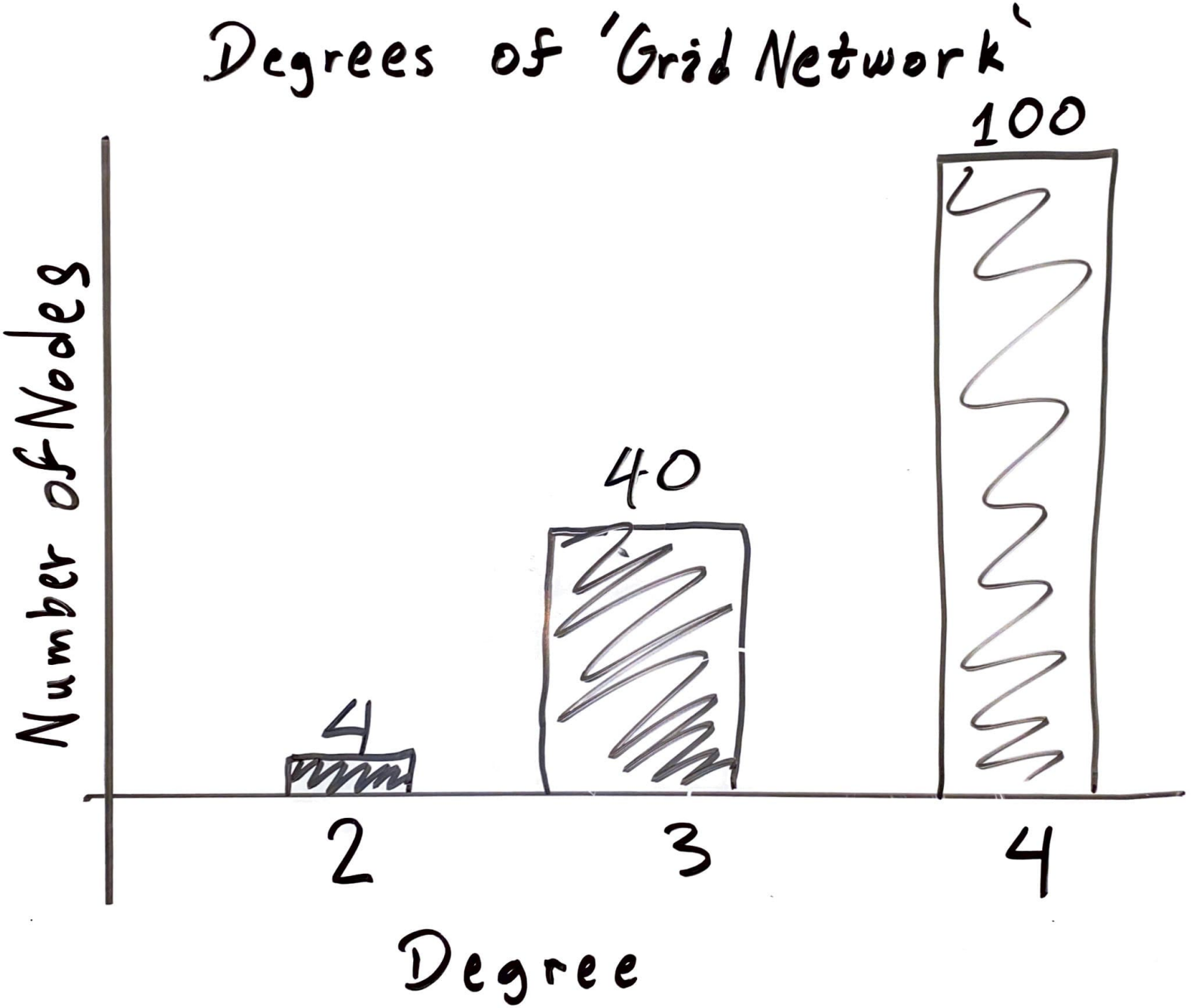
```
cutset = list()                # initialize cutset as a list
for i in sorted(cutset_trial): # loop for every entry in cutset_trial (each node in the minimum cutset)
    cutset.append(i[0])        # add value for each node in the minimum cutset

# print the size of the minimum cut set and the nodes within it
print('minimum cut set size:', len(cutset))
print('minimum cut set contains:', cutset)
```

```
minimum cut set size: 2
minimum cut set contains: ['Clean up', 'Finish valve chambers']
```

What would be a potential use-case for analyzing the minimum cut of an activity network?

a potential use-case for analysizing the minimum cut of an activity network could be preventing bottlenecking that could slow down the timeline of the project


Degrees of 'Grid Network'
100
40
4
Number of Nodes
2
3
4
Degree