CSCE 5013 - Graph Theory

Assignment 1

Zack Fravel

1 Book Exercises

1. What is the number of edges in K^n?

   - There are n vertices in K^n
   - Since K^n is fully connected, for each vertex we have (n-1) edges
   - Finally, we divide by two since each edge is connected to two vertices
   - Therefore, the Edges in K^n can be represented by:

$$\frac{n(n-1)}{2}$$

2. Let $d \in N$ and $V = (0,1)^d$, thus, V is the set of all 0 - 1 sequences of length d. The graph on V in which two such sequences form an edge if and only if they differ in exactly one position is called the d-dimensional cube. Determine the average degree, number of edges, diameter, girth, and circumference of this graph.

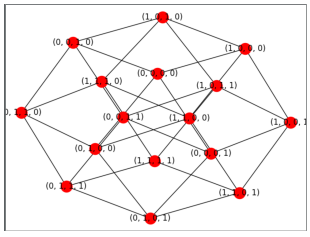

Figure 1: 4-dimensional Hyper-cube

   - The average degree of G is d. This is because each binary value associated with each vertex can only change by one for an edge to form, therefore the degree of each vertex is d because there are d values that can possibly change at any one time.
   - The average number of edges, using some of the information laid out above, would be 2^d (The number of nodes, since 2^n gives us the binary combinations of n-length) multiplied by d (number of edges for each node), all divided by 2 (to avoid double counting).

$$\frac{d(2^d)}{2}$$

   - The diameter of G, or the longest shortest path, will always be d. To see this intuitively, consider the example $d = 2$. The 'longest shortest path' from 00 to 11 is $d = 2$ steps, from 00 to 01/10 to 11. The same is true for $d = 3$, the 'longest shortest path' being $d = 3$ steps from 000 to 001/010/100 to 011/110/101 to 111. This trend continues for all values.
   - For $d \geq 2$, the girth of G, or the shortest cycle, is always 4. If you you were able to have a cycle of length three, this would mean one of the vertices would have different by two different values. You can think of this as tracing out one of the faces of the cube, which would be the smallest cycle.

- Finally, for $d \geq 2$, the circumference of G is 2^d since all vertices are fully connected there will always exists a path to connect all vertices differing by one value.

3. Show that every $2-$*connected* graph contains a cycle.

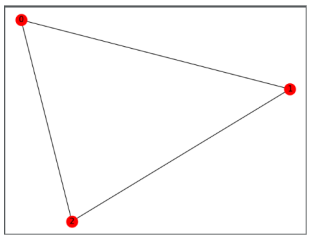

Figure 2: The simplest 2-connected graph

- Consider the simplest 2-connected graph with 3 vertices, [A, B, C].
- Say we take ${}_aP_c$, the path without B.
- Given the definition of 2-connected, ${}_cP_a$ through B still exists, thereby creating a cycle with the first path.
- No matter how many vertices you add in between A and C this will still hold true since there will always be an alternate path to every vertex (2-connected).
- Therefore, every $2-$*connected* graph contains a cycle.

4. (i) Determine $\kappa(G)$ and $\lambda(G)$ for $G = P^k$, C^k, K^k, and $K^{m,n}$

Below I have given a table of all connectivity and edge connectivity values.

$G =$	P^k	C^k	K^k	$K^{m,n}$
$\kappa(G)$	1	2	k-1	$min(m,n)$
$\lambda(G)$	1	2	k-1	$min(m,n)$

A rule to keep in mind when considering all of these definitions is $\kappa() \leq \lambda() \leq \delta()$. For paths of any length, they have edge-connectivity and connectivity of 1. This is because for both edges and vertices it only takes removing one to disconnect the graph. For a cycle, taking one edge or vertex turns it into a path, then taking a second disconnects it. For the complete graph K^k, connectivity and edge-connectivity is k-1 since the degree of every vertex is k-1 it takes removing that many edges and vertices. Finally, for the complete bipartite graph, $K^{m,n}$ both connectivity values are $min(m,n)$ since that's how many vertices and/or edges it would take to disconnect the graph.

(ii) Determine the connectivity of the n-dimensional cube.

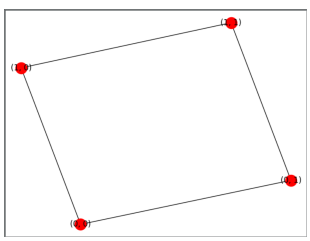

Figure 3: 2-dimensional Hyper-cube

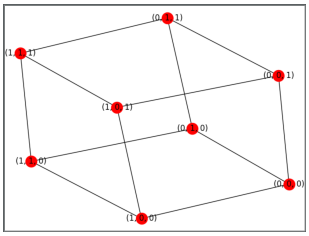

Figure 4: 3-dimensional Hyper-cube

For any n-dimensional cube, the connectivity is n. Again, it's worth remembering that $\kappa() \leq \lambda() \leq \delta()$ to help conceptualize this. The connectivity MUST be less than or equal to the minimum degree. Since for an n-dimensional cube every node is connected to n other nodes, it takes removing n nodes to disconnect any one node from the graph.

5. Let G be a connected graph, and let $r \in G$ be a vertex. Starting from r, move along the edges of G, going whenever possible to a vertex not visited so far. If there is no such vertex, go back along the edge by which the current vertex was first reached (unless the current vertex is r, then stop). Show that the edges traversed form a normal spanning tree in G with root r. (This procedure has earned those trees the name of depth-first search trees.)

   If we run the depth-first search and call the result T, we can represent the result as a subgraph of G. $T \subseteq G$. Since T never counts any vertex twice we can be sure that T is acyclic. Since G is a connected graph, we can safely say that T "spans" G. Given these two facts, we can now say that T is a spanning tree of G.

   Every time we visit a new vertex, we're increasing the "order" of the tree. At a vertex "A" the next available vertices, say "C" and "D", will be the order of "A" + 1. This is guaranteed even if we have already backtracked to "A" from a vertex "B", for instance. "B" would have also have had an order + 1 than "A" in this case. With this fact, we can now guarantee that T is a normal spanning tree of G.

6. Show that a graph is bipartite if and only if every *induced* cycle has even length.

   - $\Rightarrow$
     - Consider any bipartite graph that contains a cycle.
     - Given the nature of a bi-partitioned graph, at minimum it takes 4 steps to create a cycle.
     - Start at one node in partition A, move to another node (Partition B) , then move to another node (Partition A), then to another (Partition B), finally from here we can move back to the original node.
     - Notice this cycle is an induced subgraph of G, and this always holds true since to create a new cycle we need to create parity on both sides (one added vertex for A means one for B).
     - Therefore we can conclude that if a graph is bipartite, it contains an induced cycle with even length.
   - $\Leftarrow$
     - To prove in the other direction, we just need to show that if you have an induced cycle of odd length then you cannot bi partition the vertices.
     - Image a graph with three connected vertices, this G would contain an induced cycle of odd length.
     - Notice how it is impossible to bi partition the vertices, for example with squares and circles. To be bipartite we need to have all squares only connected to circles and all circles only connected to squares.
     - However, with induced cycles of odd length, no matter how long (you can keep adding vertices one at a time alternating square and circle and never get bipartite), you will always have connections between circles and squares if there are odd cycles.
     - Therefore, we can conclude that graphs with induced cycles of even length are also bipartite.

2 Programming Exercise

1. Explain how to use Breadth-First-Search to compute the girth of a graph. Give the run-time in O notation.

   We start taking a Breadth-First-Search for every node as the root. We add the current node to a list and check for neighbors. When we check for the neighbors, we want to check if the node is already contained in the list. If the node is already in the list, then there exists a cycle on G since the way we arrived at the current node was along a path of the vertices already in the list. If neighbors are all new, then we add them to the list and set current node to the next node on the list. Since we're looping over all the nodes and then also looping over each node to evaluate edge connections, we can represent the algorithm as...

   $$\mathcal{O}(V^2 + VE)$$

2. Describe an algorithm for finding the diameter based on a shortest path algorithm. Give the run-time for this algorithm.

   The algorithm I came up with for calculating diameter involves taking advantage of a networkx function which returns a dictionary of all the shortest paths given a root node. I iterated through all the nodes, then iterate through all possible shortest paths, comparing each to the previous shortest path. The longest of these paths is the diameter of the graph. Since the algorithm uses a nested for loop, its run-time can be represented as...

   $$\mathcal{O}(V^2)$$

3. Write a program to find the girth and diameter of the graphs given in the provided text files. Assume that all of the edge weights are 1. It should be possible to run this program from the command line with no arguments and the provided files in the same directory, and it will output each graph number along with its girth and diameter.

```
# --------------------------------------
# Zack Fravel
# 010646947
# Homework 1
# CSCE 5014 Applications of Graph Theory
# --------------------------------------
# HW1.py
#
# Read in multiple graphs and calculate
# their girth and diameter.
# --------------------------------------

import networkx as nx

# Read in undirected graphs from .txt files using nx.read_edgelist function
G1 = nx.read_edgelist("graph1.txt")
G2 = nx.read_edgelist("graph2.txt")
G3 = nx.read_edgelist("graph3.txt")
G4 = nx.read_edgelist("graph4.txt")


# Algorithm for finding the girth using a breadth-first-search
def girth(G):
    # Create list of nodes to search through and empty list of checked nodes
    toCheck = list(map(int,G.nodes()))
```

```
    checked = []
    checker = 0
    current_cycle = 10000000
    short_cycle = 10000000

    # perform this through the each node
    for n in G.nodes():
        # set the current node to toCheck[checker]
        # this checker value will increase on each iteration of the loop
        current_node = toCheck[checker]
        # add current_node to checked[] list
        checked.append(current_node)
        bounds = checked.__len__()
        neighbors = list(map(int,G.neighbors(n)))
        # Loop over all the neighbors
        for adj in neighbors:
            i = 0
            while i < bounds:
                if adj == checked[i]:
                    # Check for disconnected graph
                    # Check if a path exists between the current_node and root
                    if nx.has_path(G, '%s' % current_node, '%s' % 0 ):
                        current_cycle = nx.shortest_path_length(G, '%s' % current_node, '%s' % 0)
                    else:
                        print("Graph is disconnected")
                    # This solves the case for a bipartite graph, since the shortest path will always
                    # In a bipartite graph the shortest cycle is always 3
                    if G.degree('%s' % current_node) == len(G) - 1:
                        short_cycle = 3
                    # Invalidates paths of length 1 since it needs to be at least 2 away from the roo
                    if current_cycle > 1:
                        current_cycle = current_cycle + 1
                        # If current_cycle is shorter than the previous short cycle, set new short_cy
                        if current_cycle < short_cycle:
                            short_cycle = current_cycle
                i = i + 1
        checker = checker + 1

    if(short_cycle == 10000000):
        print("There are no cycles, Graph has infinite girth.")
    else:
        print("Girth = ", short_cycle)


# Algorithm for finding the "Longest Shortest Path" or Diameter of G
def diameter(G):
    # Set original diameter to 0
    diam = 0
    # check for all nodes
    for n in G.nodes():
        # find the shortest path across the graph
        d = nx.shortest_path_length(G, n)
        # check for all paths on that node
        for key in d.keys():
            # if new path is longer than old path, then this is the new diameter
            if d[key] > diam:
                diam = d[key]
```

```
    # Print diameter
    print("Diameter = ", diam, "\n\n")


print("Graph 1")
girth(G1)
diameter(G1)

print("Graph 2")
girth(G2)
diameter(G2)

print("Graph 3")
girth(G3)
diameter(G3)

print("Graph 4")
girth(G4)
diameter(G4)
```

```
OUTPUT:

/Users/Meryl/PycharmProjects/Workspace/venv/bin/python /Users/Meryl/PycharmProjects/Graphs/hw1/HW1.py
Graph 1
Girth =  3
Diameter =  3


Graph 2
Girth =  3
Diameter =  1


Graph 3
Graph is disconnected
Girth =  3
Diameter =  10


Graph 4
Girth =  3
Diameter =  3



Process finished with exit code 0
```