Application 2: Analysis of a Computer Network
Algorithmic Thinking (Part1)

Shamsuddin Rehmani

July 16, 2019

Application 2 Description

Graph exploration (that is, “visiting” the nodes and edges of a graph) is a powerful and necessary tool to elucidate properties of graphs and quantify statistics on them. For example, by exploring a graph, we can compute its degree distribution, pairwise distances among nodes, its connected components, and centrality measures of its nodes and edges. As we saw in the Homework and Project, breadth-first search can be used to compute the connected components of a graph.

In this Application, we will analyze the connectivity of a computer network as it undergoes a cyber-attack. In particular, we will simulate an attack on this network in which an increasing number of servers are disabled. In computational terms, we will model the network by an undirected graph and repeatedly delete nodes from this graph. We will then measure the resilience of the graph in terms of the size of the largest remaining connected component as a function of the number of nodes deleted.

Answer to Question 1

Probability p such that the ER graph computed using this edge probability has approximately the same number of edges as the computer network is 0.00397

Integer m such that the number of edges in the UPA graph is close to the number of edges in the computer network is 2

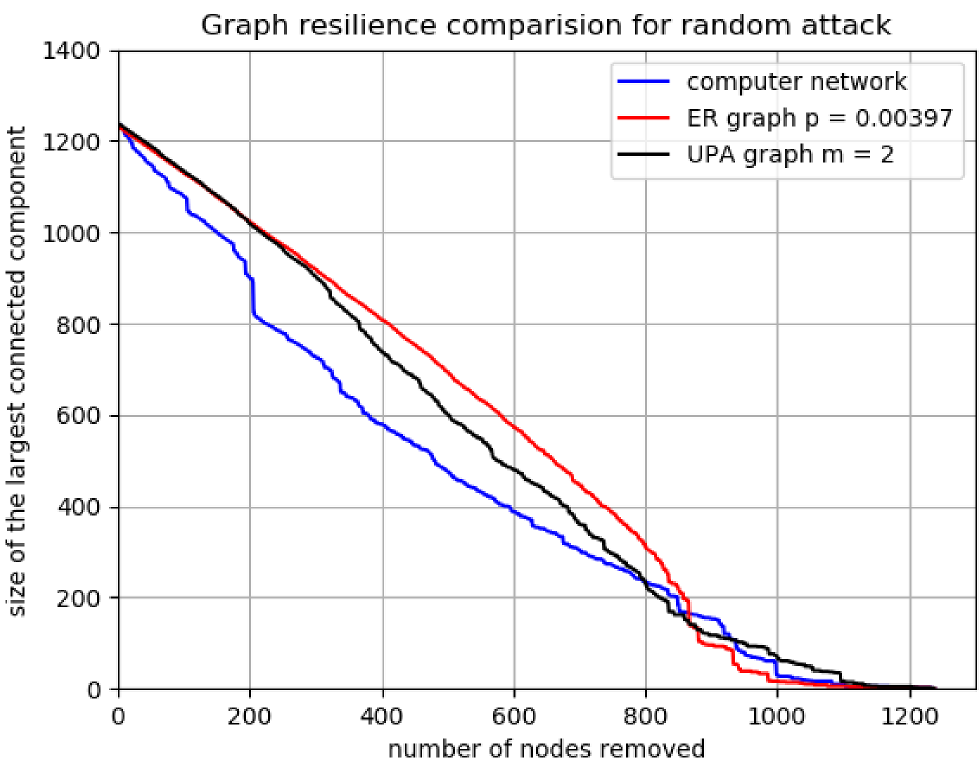

Figure 1: graph resilience comparision for random attack for computer network, undirected ER graph, and UPA graph

Answer to Question 2

All 3 graphs seem to be resilient (as shown in Figure 1), i.e the size of the largest connected component is within 25% of 1000 (the approximate number of remaing nodes)

Answer to Question 3

Big-O bounds for `target_order` $= O(n^2 + m) = O(n^2)$ since for UPA graph $m <= 5n$ (m = total number of edges in UPA)

Big-O bounds for `fast_target_order` $= O(n + m) = O(n)$ since for UPA graph $m <= 5n$ (m = total number of edges in UPA)

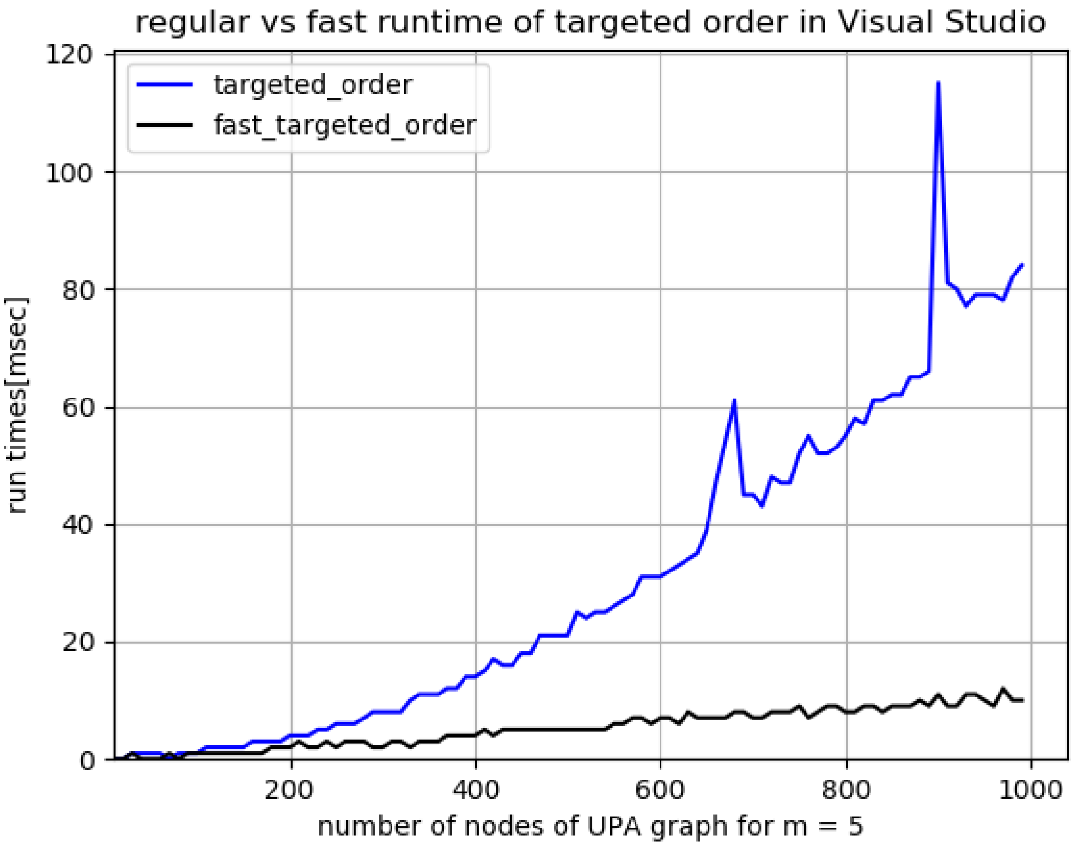

Figure 2: runtime for `target_order` and `fast_target_order` on UPA graph with nodes ranging from 10 to 1000 for $m = 5$

Answer to Question 4

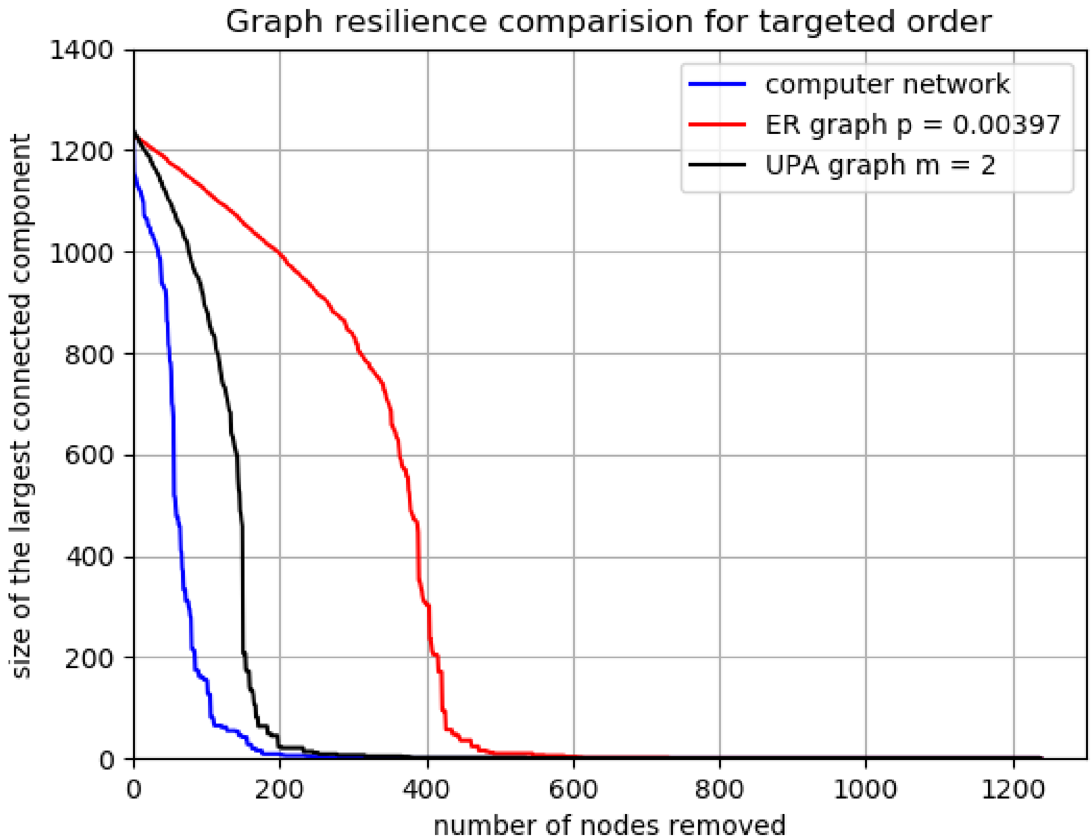

Figure 3: graph resilience comparision for targeted attack for computer network, undirected ER graph, and UPA graph

Answer to Question 5

From the graph(Figure 3) we can see that only ER graph is resilient as the first 20% of the nodes are removed while the UPA and the computer network graph reaches close to zero as 20% of the ndoes are removed

A Python code used to answer the Application Questions

```python
"""
Solutiion for Application #2: "Analysis of a Computer Network"
"""

import time
import random
import matplotlib.pyplot as plt
import numpy as np
import alg_application2_provided as alg_app2_prov
import alg_example_graphs as alg_graphs
import alg_project2_solution as alg_proj2_sol

##### Q1 Solution #####
# To begin our analysis, we will examine the resilience of the computer network under
# an attack in which servers are chosen at random. We will then compare the resilience
# of the network to the resilience of ER and UPA graphs of similar size.
#
# To begin, you should determine the probability pp such that the ER graph computed
# using this edge probability has approximately the same number of edges as the computer
# network. (Your choice for pp should be consistent with considering each edge in the
# undirected graph exactly once, not twice.) Likewise, you should compute an integer mm
# such that the number of edges in the UPA graph is close to the number of edges in the
# computer network. Remember that all three graphs being analyzed in this Application
# should have the same number of nodes and approximately the same number of edges.

# load the graph from a text file
cnet_graph = alg_app2_prov.load_graph(alg_app2_prov.NETWORK_URL)

# get the number of nodes in the computer network graph
num_nodes = len(cnet_graph.keys())

# find the total number of edges in th computer network graph
edges = sum([len(neighbors) for  neighbors in cnet_graph.values()])/2

# find the probability such that the ER graph computed using this edge
# probability has approximately the same number of edges as the computer network
prob_p = round(2.0 * edges / (num_nodes * (num_nodes - 1.0)), 6)

# get the average degree so that the graph created using UPA algorithm has approximately
# same number of edges as network_graph
m_nodes = int(round(float(edges)/num_nodes))

# generate the random graph based on ER algorithm
er_graph = alg_graphs.alg_er(num_nodes, prob_p)

# generate the random graph based on UPA algorithm
upa_graph = alg_graphs.alg_upa(num_nodes, m_nodes)

# Next, you should write a function random_order that takes a graph and returns a list
# of the nodes in the graph in some random order. Then, for each of the three graphs
# (computer network, ER, UPA), compute a random attack order using random_order and use
# this attack order in compute_resilience to compute the resilience of the graph.

def random_order(graph):
    """
    Take a graph a returns a random sequence of its nodes
    Arguments:
        graph {dictionary} -- [a graph]

    Returns:
        list of nodes -- random sequence of nodes
    """

    lst_nodes = graph.keys()
    random.shuffle(lst_nodes)

    return lst_nodes

# compute the resilience of each of the 3 graphs
cnet_res = alg_proj2_sol.compute_resilience(cnet_graph, random_order(cnet_graph))
er_res = alg_proj2_sol.compute_resilience(er_graph, random_order(er_graph))
upa_res = alg_proj2_sol.compute_resilience(upa_graph, random_order(upa_graph))

# Once you have computed the resilience for all three graphs, plot the results as three
# curves combined in a single standard plot (not log/log). Use a line plot for each curve.
# The horizontal axis for your single plot be the the number of nodes removed (ranging
# from zero to the number of nodes in the graph) while the vertical axis should be the
# size of the largest connect component in the graphs resulting from the node removal.
# For this question (and others) involving multiple curves in a single plot, please
# include a legend in your plot that distinguishes the three curves. The text labels in
# this legend should include the values for pp and mm that you used in computing the ER
# and UPA graphs, respectively. Both matplotlib and simpleplot support these capabilities
# (matplotlib example and simpleplot example).
#
# Note that three graphs in this problem are large enough that using CodeSkulptor to
# calculate compute_resilience for these graphs will take on the order of 3-5 minutes
# per graph. When using CodeSkulptor, we suggest that you compute resilience for each
# graph separately and save the results (or use desktop Python for this part of the
# computation). You can then plot the result of all three calculations using simpleplot.
# load the graph from the text file

# compute the list of number of nodes removed (ranging from zero to the number of nodes in the graph)
num_removed = range(num_nodes + 1)
```

```python
# plot the graphs of resilience vs number of nodes removed for each of the 3 graphs
plt.figure(0)
plt.plot(num_removed, cnet_res, '-b', label = 'computer network')
plt.plot(num_removed, er_res, '-r', label = 'ER graph p = 0.00397')
plt.plot(num_removed, upa_res, '-k', label = 'UPA graph m = 2')
plt.title('Graph resilience comparision for random attack')
plt.xlabel('number of nodes removed')
plt.ylabel('size of the largest connected component')
plt.legend(loc = 'upper right')
plt.xlim(0, None)
plt.ylim(0, 1400)
plt.grid()
# uncommet to save the plot
#plt.savefig("Q1_graph_resilience_comparision.png")

##### Q2 Solution #####
# Consider removing a significant fraction of the nodes in each graph
# using random_order. We will say that a graph is resilient under this
# type of attack if the size of its largest connected component is
# roughly (within ~25%) equal to the number of nodes remaining, after
# the removal of each node during the attack.
#
# Examine the shape of the three curves from your plot in Question 1.
# Which of the three graphs are resilient under random attacks as the
# first 20% of their nodes are removed?
#
# Ans: all 3 graphs seem to be resilient, i.e the size of the largest
# connected component is within 25% of 1000 ( the approximate number of
# remaing nodes)

##### Q3 Solution #####
# In the next three problems, we will consider attack orders in which the
# nodes being removed are chosen based on the structure of the graph. A
# simple rule for thesetargeted attacks is to always remove a node of
# maximum (highest) degree from the graph. The function targeted_order(ugraph)
# in the provided code takes an undirected graph ugraph and iteratively does
# the following:
#
# - Computes a node of the maximum degree in ugraph. If multiple nodes have
#   the maximum degree, it chooses any of them (arbitrarily).
# - Removes that node (and its incident edges) from ugraph.
#
# Observe that targeted_order continuously updates ugraph and always computes
# a node of maximum degree with respect to this updated graph. The output of
# targeted_order is a sequence of nodes that can be used as input to compute_resilience.
#
# As you examine the code for targeted_order, you feel that the provided
# implementation of targeted_order is not as efficient as possible. In
# particular, much work is being repeated during the location of nodes
# with the maximum degree. In this question, we will consider an alternative
# method (which we will refer to as fast_targeted_order) for computing the
# same targeted attack order. In Python, this method creates a list
# degree_sets whose kth element is the set of nodes of degree k. The method
# then iterates through the list degree_sets in order of decreasing degree.
# When it encounter a non-empty set, the nodes in this set must be of
# maximum degree. The method then repeatedly chooses a node from this set,
# deletes that node from the graph, and updates degree_sets appropriately.
#
# For this question, your task is to implement fast_targeted_order and then
# analyze the running time of these two methods on UPA graphs of size n with
# m = 5.

# Determine big-O bounds of the worst-case running times of targeted_order
# and fast_targeted_order as a function of the number of nodes n in the UPA graph.
# Since the number of edges in these UPA graphs is always less than 5n (due to the
# choice of m = 5), your big-O bounds for both functions should be expressions in n.
# You should also assume that the all of the set operations used in fast_targeted_order
# are O(1).
#
# Ans: target_order = O(n^2 + m) = O(n^2) since for UPA graph m <= 5n (m = total
#      number of edges in UPA)
#      fast_target_order = O(n + m) = O(n) since for UPA graph m <= 5n (m = total
#      number of edges in UPA)

# Next, run these two functions on a sequence of UPA graphs with n in range(10,1000,10)
# and m=5 and use the time module (or your favorite Python timing utility) to compute
# the running times of these functions. Then, plot these running times (vertical axis)
# as a function of the number of nodes n (horizontal axis) using a standard plot
# (not log/log). Your plot should consist of two curves showing the results of your
# timings. Remember to format your plot appropriately and include a legend. The title
# of your plot should indicate the implementation of Python (desktop Python vs. CodeSkulptor)
# used to generate the timing results.

def fast_targeted_order(graph):
    """
    Compute a targeted attack order consisting of nodes of
    maximal degree. The algorithm used was provided.

    Arguments:
        graph {dictionary} -- a graph

    Returns:
        list of nodes -- a list of nodes of attack order with maximal
                         degree in descending order
    """

```

```python
    # intialize the list with the target node order
    node_order = []

    # make a copy of the node
    graph_cpy = alg_app2_prov.copy_graph(graph)

    # get all the nodes of the graph
    nodes = graph_cpy.keys()

    # initialize all degree set so that all degree corresponds to empty set of nodes
    degree_set = [set() for dummy_idx in nodes]

    # add all the nodes to their corresponding degree location
    for node in nodes:
        degree = len(graph_cpy[node])
        degree_set[degree].add(node)

    # update the degree set and delete the node of the copy of graph appropriately after
    # storing the node with current maximal degree
    for deg_set_idx in range(len(nodes) - 1, -1, -1):
        while(len(degree_set[deg_set_idx]) != 0):
            node_u = degree_set[deg_set_idx].pop()
            for neighbor in graph_cpy[node_u]:
                degree_neigbor = len(graph_cpy[neighbor])
                degree_set[degree_neigbor].remove(neighbor)
                degree_set[degree_neigbor - 1].add(neighbor)

            node_order.append(node_u)
            alg_app2_prov.delete_node(graph_cpy, node_u)


    return node_order

# intialize the fast and normal function times for targeted order
time_fast_targeted_order = []
time_targeted_order = []

# intialize the nodes to be used to generate the UPA graphs
nodes = range(10, 1000, 10)

# calcualte the time to run the normal and fast functions for the target order
for node in nodes:
    # create the UPA graph with n = node and m = 5 where m is the number of
    # existing nodes to which a new node is connected during each iteration
    upa_graph_new = alg_graphs.alg_upa(node, 5)
    # calculate the attack order based on normal targeted order function
    # and store the time it takes to run this function
    start = time.time()
    alg_app2_prov.targeted_order(upa_graph_new)
    end = time.time()
    time_targeted_order.append((end - start) * 1000)
    # calculate the attack order based on fast targeted order function
    # and store the time it takes to run this function
    start = time.time()
    fast_targeted_order(upa_graph_new)
    end = time.time()
    time_fast_targeted_order.append((end - start) * 1000)

# plot the graphs of resilience vs number of nodes removed for each of the 3 graphs
plt.figure(1)
plt.plot(nodes, time_targeted_order, '-b', label = 'targeted_order')
plt.plot(nodes, time_fast_targeted_order, '-k', label = 'fast_targeted_order')
plt.title('regular vs fast runtime of targeted order in Visual Studio' )
plt.xlabel('number of nodes of UPA graph for m = 5')
plt.ylabel('run times[msec]')
plt.legend(loc = 'upper left')
plt.xlim(10, None)
plt.ylim(0, None)
plt.grid()
# uncommet to save the plot
#plt.savefig("Q3_targeted_order_time_comparision.png")

##### Q4 Solution #####
# To continue our analysis of the computer network, we will examine its resilience
# under an attack in which servers are chosen based on their connectivity. We will
# again compare the resilience of the network to the resilience of ER and UPA graphs
# of similar size.
#
# Using targeted_order (or fast_targeted_order), your task is to compute a targeted
# attack order for each of the three graphs (computer network, ER, UPA) from Question 1.
# Then, for each of these three graphs, compute the resilience of the graph using
# compute_resilience. Finally, plot the computed resiliences as three curves (line plots)
# in a single standard plot. As in Question 1, please include a legend in your plot that
# distinguishes the three plots. The text labels in this legend should include the values
# for p and m that you used in computing the ER and UPA graphs, respectively.

# compute the target order for the 3 graph in Q1
tar_order_cnet = fast_targeted_order(cnet_graph)
tar_order_er = fast_targeted_order(er_graph)
tar_order_upa = fast_targeted_order(upa_graph)

# compute the resilience for the 3 graph using the targeted order
# compute the resilience of each of the 3 graphs
cnet_res = alg_proj2_sol.compute_resilience(cnet_graph, tar_order_cnet)
er_res = alg_proj2_sol.compute_resilience(er_graph, tar_order_er)
upa_res = alg_proj2_sol.compute_resilience(upa_graph, tar_order_upa)
```

```
# plot the computer resilience for the targeted order for the 3 graph
plt.figure(2)
plt.plot(num_removed, cnet_res, '-b', label = 'computer network')
plt.plot(num_removed, er_res, '-r', label = 'ER graph p = 0.00397')
plt.plot(num_removed, upa_res, '-k', label = 'UPA graph m = 2')
plt.title('Graph resilience comparision for targeted order')
plt.xlabel('number of nodes removed')
plt.ylabel('size of the largest connected component')
plt.legend(loc = 'upper right')
plt.xlim(0, None)
plt.ylim(0, 1400)
plt.grid()
plt.show()
# uncommet to save the plot
#plt.savefig("Q4_graph_resilience_comparision.png")

##### Q5 Solution #####
# Examine the shape of the three curves from your plot in Question 4.
# Which of the three graphs are resilient under targeted attacks as
# the first 20% of their nodes are removed? Again, note that there is
# no need to compare the three curves against each other in your answer
# to this question.
#
# Ans: From the graph we can see that only ER graph is resilient as the
# first 20% of the nodes are removed while the UPA and the computer
# network graph reaches close to zero as 20% of the ndoes are removed
```

B All functions for project 4 used in the application

```
"""
Functions for Project #2: "Connected Components and Graph Resilience". These
functions will be used for the Application #2: "Analysis of a Computer Network"
"""
from collections import deque
import alg_application2_provided as alg_app2_prov

def bfs_visited(ugraph, start_node):
    """
    Perfoms the Breath First Seach(BFS) and returns a set of all the nodes
    that are visited starting from start_node

    Arguments:
        ugraph {dictionary} -- an undirect graph
        start_node {integer} -- a node in the ugraph

    Returns:
        set -- a set of all nodes visited by a BFS that start at start_node
    """
    # initialize an queue with the start_node. We use python's built in double
    # ended queue, deque
    deq = deque([start_node])
    # add the start node to the visited nodes set
    visited = set([start_node])
    # keep traversing through all the neighbors of the nodes in the queue
    # as long as the queue is not empty and mark them as visited if the nodes
    # are not yet visited
    while len(deq) != 0:
        curr_node = deq.popleft()
        for neighbor in ugraph[curr_node]:
            if not (neighbor in visited):
                visited.add(neighbor)
                deq.append(neighbor)

    return visited

def cc_visited(ugraph):
    """
    Takes an undirected graph ugraph and computes the all the
    connected components of the graph

    Arguments:
        ugraph {dictionary} -- an undirected graph

    Returns:
        list of sets -- resturns a list of sets where each set has all the nodes in
                        a particular connected component of the graph, and each set
                        represent a connect component of the graph
    """
    # initialize the remaining nodes in the ugraph that have not yet been visited
    remain_nodes = set(ugraph.keys())
    # initiakize the list of sets where each set is a connected component of ugraph
    con_comp = []

    # use BFS to find all the connect components until all the nodes of the ugraph
    # have been visited
    while len(remain_nodes) != 0:
        not_vis_node = remain_nodes.pop()
        visited = bfs_visited(ugraph, not_vis_node)
        con_comp.append(visited)
        remain_nodes -= visited

    return con_comp

```

```
def largest_cc_size(ugraph):
    """
    Takes a undirected graph and returns the size of the largest connected component

    Arguments:
        ugraph {dictionary} -- an undirected graph

    Returns:
        integer -- the size of the largest connected component of ugraph
    """
    # find the size of all the connect components of the ugraph
    len_cc = [len(con_comp) for con_comp in cc_visited(ugraph)]

    # make sure to take care of the case when ugraph is empty and we get
    # an empty len_cc list
    if (len(len_cc) == 0):
        return 0

    # return the max size of connected compo
    return max(len_cc)

def compute_resilience(ugraph, attack_order):
    """
    Computes a measure of resilience of an undirected graph. Takes the undirected
    graph ugraph, a list of nodes attack_order and iterates through the nodes in
    attack_order. For each node in the list, the function removes the given node
    and its edges from the graph and then computes the size of the largest connected
    component for the resulting graph.

    Arguments:
        ugraph {dictionary} -- an undirected graph
        attack_order {list of nodes} -- list of nodes that will be iterated over

    Returns:
        list of integers -- return a list whose (k+1)th entry is the size of the largest
                            connected component in the graph after the removal of the first
                            k nodes in attack_order
    """
    new_graph = alg_app2_prov.copy_graph(ugraph)

    # get the size of the largest connected component before removing any nodes
    lst_max_cc = [largest_cc_size(new_graph)]

    # start removing each node in the attack_order and its edges from the ugraph
    # and find the largest connected component after each removal
    for remove_node in attack_order:
        alg_app2_prov.delete_node(new_graph, remove_node)
        lst_max_cc.append(largest_cc_size(new_graph))

    return lst_max_cc
```

C Code for ER and UPA graph generations

```
"""
Funtions to generate 2 types of ugraphs, the undirected ER graph
and UPA graph
"""
import random
import alg_upa_trial as upa

def make_complete_graph(num_nodes):
    """
    create and return a complete graph with nodes from
    0 to num_nodes - 1 for num_nodes > 0. Otherwise
    the function returns a dictionary corresponding to
    the empty graph

    Arguments:
        num_nodes {integer} -- number of nodes for the graph

    Returns:
        dictionary -- returns a dictionary corresponding to a complete directed
        graph with the specified number of nodes.
    """
    # local variable for the complete graph
    graph = {}

    # return an empty graph if num_nodes is not positive
    if num_nodes <= 0:
        return graph

    for node in range(num_nodes):
        # create an adjacency list for a directed complete graph with no
        # self loops or parallel edges
        graph[node] = set([val for val in range(num_nodes) if val != node])

    return graph


def alg_er(num_nodes, p):
    """
    generate a random graph based on Erdos Renyi(ER) model G(n, p)
    where each edge in the graph is added with probability p

```

```python
    Arguments:
        num_nodes {integer} -- the total number of nodes for the generated graph
        p {float} -- the probability with which to add each edge to the generated graph

    Returns:
        dictionary -- return the ER random graph
    """

    ugraph = {}
    all_edges = []

    if (num_nodes <= 0):
        return ugraph

    # create a graph of all nodes but no edges
    for node in range(num_nodes):
        ugraph[node] = set()

    # find all possible edges of the graph
    all_edges = (set([frozenset([node1, node2]) for node1 in range(num_nodes)
        for node2 in range(num_nodes) if node1 != node2]))

    # convert each edge from frozenset to list for indexing
    all_edges = [list(item) for item in all_edges]

    # add edges to the graph with probablity p
    for edge in all_edges:
        rand_prob = random.uniform(0,1)
        if rand_prob < p:
            ugraph[edge[0]].add(edge[1])
            ugraph[edge[1]].add(edge[0])

    return ugraph

def alg_upa(n_nodes, m_nodes):
    """
    Uses the DPA algorithm provided in Q3 of the Application #1
    to generates a random undirected graph iteratively, where
    each iteration a new node is created, added to the graph,
    and connected to the subset of the existing node

    Arguments:
        n_nodes {integer} -- final number of nodes in the generated graph
        m_nodes {integer} -- number of existing nodes to which a new node is connected
                             during each iteration

    Returns:
        dictionary -- the generated graph based on modified DPA algorithm for undirectef graph
    """
    # create a complete graph of m_nodes noes
    upa_graph = make_complete_graph(m_nodes)

    # create the UPAtrial object corresponding to complete graph
    upa_trial = upa.UPATrial(m_nodes)

    # add each new node to the existing graph randomly
    # chosen with probability:
    # (in-degree of new_node + 1) / (in-degree of all nodes +
    # total number of existing nodes)
    # simulated by the run_trial of the UPATrial class
    for new_node in range(m_nodes, n_nodes):
        # randomly select m_nodes from the existing graph that
        # the new_node will be connected to. Remove if any
        # duplicate nodes in the m_nodes selected
        neighbors = upa_trial.run_trial(m_nodes)

        # update the existing graph to add this new node and its
        # neighbors
        upa_graph[new_node] = neighbors

        # add this new node to all the neighbor nodes since this
        # is a undirected graph
        for neighbor in neighbors:
            upa_graph[neighbor].add(new_node)

    return upa_graph
```