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.

Example graphs

In this Application, you will compute the resilience of several types of undirected graphs. We suggest that you begin by collecting and writing code to create the following three types of graphs:

- **An example computer network** - The text representation for the example network is here. You may use this provided code to load the file as an undirected graph (with 1239 nodes and 3047 edges). Note that this provided code also includes several useful helper functions that you should review.
- **ER graphs** - If you have not already implemented the pseudo-code for creating *undirected* ER graphs, you will need to implement this code. You may wish to modify your implementation of `make_complete_graph` from Project 1 to add edges randomly (again, keep in mind that in Project 1 the graphs were directed, and here we are dealing with undirected graphs).
- **UPA graphs** - In Application 1, you implemented pseudo-code that created DPA graphs. These graphs were directed (The D in DPA stands for directed). In this Application, you will modify this code to generate undirected UPA graphs. The UPA version should add an undirected edge to the UPA graph whenever you added a directed edge in the DPA algorithm. Note that since the degree of the newly added node is no longer zero, its chances of being selected in subsequent trials increases. In particular, you should either modify the `DPATrial class` to account for this change or use our provided UPATrial class.

Question 1 (5 pts)

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 p such that the ER graph computed using this edge probability has approximately the same number of edges as the computer network. (Your choice for p should be consistent with considering each edge in the undirected graph exactly once, not twice.) Likewise, you should compute an integer m 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.

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.

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 p and m 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`.

Once you are satisfied with your plot, upload your plot into the peer assessment. Your plot will be assessed based on the answers to the following three questions:

- Does the plot follow the formatting guidelines for plots?
- Does the plot include a legend? Does this legend indicate the values for p and m used in ER and UPA, respectively?
- Do the three curves in the plot have the correct shapes?

Question 2 (1 pt)

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? Note that there is no need to compare the three curves against each other in your answer to this question.

Question 3 (3 pts)

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 these *targeted 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. Here is a pseudo-code description of the method:

Algorithm 1: FastTargetedOrder.

Input: Graph $g = (V, E)$, with $V = \{0, 1, \ldots, n-1\}$.
Output: A (ordered) list L of the nodes in V in decreasing order of their degrees.

```
for k ← 0 to n − 1 do
    DegreeSets[k] ← ∅;      // DegreeSets[k] is a set of all nodes whose degree is k
for i ← 0 to n − 1 do
    d ← degree(i);                                   // d is the degree of node i
    DegreeSets[d] ← DegreeSets[d] ∪ {i};
L ← [];                                        // L is initialized to an empty list
i ← 0;
for k ← n − 1 downto 0 do
    while DegreeSets[k] ≠ ∅ do
        Let u be an arbitrary element in DegreeSets[k];
        DegreeSets[k] ← DegreeSets[k] − {u};
        foreach neighbor v of u do
            d ← degree(v);
            DegreeSets[d] ← DegreeSets[d] − {v};
            DegreeSets[d − 1] ← DegreeSets[d − 1] ∪ {v};
        L[i] ← u;
        i ← i + 1;
        Remove node u from g;
return L;
```

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 $m = 5$. Your analysis should be both mathematical and empirical and include the following:

- 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.
- Compute a plot comparing the running times of these methods on UPA graphs of increasing size.

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 $O(1)$.

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.**

Your answer to this question will be assessed according to the following three items:

- What are tight upper bounds on the worst-case running times of `targeted_order` and `fast_targeted_order`? Use big-O notation to express your answers (which should be very simple).
- Does the plot follow the formatting guidelines for plots? Does the plot include a legend? Does the title include the implementation of Python used to compute the timings?
- Are the shapes of the timing curves in the plot correct?

Question 4 (5 pts)

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.

Once you are satisfied with your plot, upload your plot into the peer assessment. Your plot will be assessed based on the answers to the following three questions:

- Does the plot follow the formatting guidelines for plots?
- Does the plot include a legend? Does this legend indicate the values for p and m used in ER and UPA, respectively?
- Do the three curves in the plot have the correct shape?

Question 5 (1 pt)

Now, consider removing a significant fraction of the nodes in each graph using `targeted_order`. 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.

Question 6 (optional, no credit)

An increasing number of people with malevolent intent are interested in disrupting computer networks. If you found one of the two random graphs to be more resilient under targeted attacks than the computer network, do you think network designers

should always ensure that networks' topologies follow that random model? Think about the considerations that one might have to take into account when designing networks and provide a short explanation for your answer.

Marquer comme terminé