coursera

Project #2 Description

For the Project component of Module 2, you will first write Python code that implements breadth-first search. Then, you will use this function to compute the set of connected components (CCs) of an undirected graph as well as determine the size of its largest connected component. Finally, you will write a function that computes the resilience of a graph (measured by the size of its largest connected component) as a sequence of nodes are deleted from the graph.

You will use these functions in the Application component of Module 2 where you will analyze the resilience of a computer network, modeled by a graph. As in Module 1, graphs will be represented using dictionaries.

Breadth-first search

For this part of Project 2, you will implement the pseudo-code for **BFS-Visited**from question 13 in Homework 2. In particular, your task is to implement the following function:

- `bfs_visited(ugraph, start_node)` - Takes the undirected graph `ugraph` and the node `start_node` and returns the set consisting of all nodes that are visited by a breadth-first search that starts at `start_node`.

To implement the queue used in BFS, we recommend one of two options. For desktop Python users, you may use the `collections.deque` module which supports $O(1)$ enqueue and dequeue operations. Use the statement `from collections import deque` to import this module for use with OwlTest. Note that using the list operations `pop(0)` and `append(...)` to model enqueue and dequeue will lead to a slower implementation since popping from the front of a list is $O(n)$ in desktop Python.

For CodeSkulptor users, the list operations `pop(0)` and `append(...)` both have fast implementations in CodeSkulptor. As a result, you are welcome to either import the `poc_queue` module provided for "Principles of Computing" (via the statement `import poc_queue`) or directly implement a queue using the list operations `pop(0)` and `append(...)`.

Connected components

For this part of Project 2, you will implement the pseudo-code for**CC-Visited** from question 13 in Homework 2. In particular, your task is to implement the following two functions:

- `cc_visited(ugraph)` - Takes the undirected graph `ugraph` and returns a list of sets, where each set consists of all the nodes (and nothing else) in a connected component, and there is exactly one set in the list for each connected component in `ugraph` and nothing else.
- `largest_cc_size(ugraph)` - Takes the undirected graph `ugraph` and returns the size (an integer) of the largest connected component in `ugraph`.

To ensure an efficient implementation of `cc_visited`, we recommend that you use the function `bfs_visited` in implementing `cc_visited`.

Graph resilience

In the Application component of this Module, we will study the connectivity of computer networks. In particular, we will subject a model of one particular network to random and targeted "attacks". These attacks correspond to disabling a sequence of servers in the network and will be simulated by removing a sequence of nodes in the graph that corresponds to these servers.

For this part of the Project, your task is to implement a function that takes an undirected graph and a list of nodes that will be attacked. You will remove these nodes (and their edges) from the graph one at a time and then measure the "resilience" of the graph at each removal by computing the size of its largest remaining connected component. In particular, your task is to implement the function:

- **`compute_resilience(ugraph, attack_order)`** - 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. The function should 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`**. The first entry (indexed by zero) is the size of the largest connected component in the original graph.

The easiest method for implementing **`compute_resilience`** is to remove one node at a time and use **`largest_cc_size`** to compute the size of the largest connected component in the resulting graphs. This implementation has a running time of $O(n(n+m))$ where n is the number of nodes and m is the number of edges in the graph.

Challenge problem: In the Application, you will compute the resilience of graphs with several thousand nodes and edges. In desktop Python, this computation will take on the order of a few seconds. In CodeSkulptor, this computation will take 3-5 minutes per graph. As a challenge, investigate other asymptotically faster approaches to implementing **`compute_resilience`**. We have implemented one approach based on a simple disjoint set algorithm that has a running time of $O(n\log(n)+m)$. This method can perform all of the analysis in the Application part of this Module in CodeSkulptor in a few seconds.

Please note that we **strongly recommend** that you use **`largest_cc_size`** in your first implementation of **`compute_resilience`** when completing the Project and Application. If you wish to use CodeSkulptor, but would like to avoid the 3-5 minute wait times, you can compute the resiliences in IDLE and paste the resulting data back into CodeSkulptor for plotting.

Grading and coding standards

As you implement each function, remember to test that function thoroughly using test data of your creation. Once you are confident that your implementation is correct, submit your code to this Owltest page. This page will automatically test your project. The example graphs used in OwlTest are available here.

OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums.

When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this project that is linked on the main programming assignment page. Remember that submitting to OwlTest does not record a grade for the assignment.

Mark as completed