Lab 7: Faster Gift Delivery

Table of Contents

1) Preparation

This lab assumes you have Python 3.5 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab7.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- correctly answering the questions on this page (0.5 points)
- passing the test cases from `test.py` under the time limit (1.5 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (2 points).

For this lab, you will only receive credit for a test case if it runs to completion under the time limit on the server.

2) Introduction

Santa has made great use of the implementations of `Graph` from the previous lab. However, it turns out that Santa forgot to deliver gifts in a small country in the Balkans and time is running out! You need to create a new implementation of `Graph` that can respond to queries much faster, so that Santa can find buildings and allocate delivery teams in time for Christmas.

In this lab, you will write a new implementation of `Graph` with a much faster `query` method by first improving the general algorithm for finding pattern matches, then making use of redundancy and specialized algorithms to tackle special cases.

To begin, copy your implementation of `GraphFactory` from Lab 6 into `lab.py`, since our tests will use that to create new instances. You will need to implement the required functions in the `FastGraph` class provided in `lab.py`, but read section 3) before implementing `query`. We have copied over all methods of the `Graph` interface (in `graph.py`) for convenience. We leave the base internal representation of `FastGraph` up to you; the additions described in section 4) should work with pretty much anything.

3) A Faster Query

For Lab 6, we asked you to implement a "brute-force" approach to `query`, but this week you will write a smarter algorithm that avoids unnecessary work. The idea is that instead of generating permutations then checking for correctness, you can "follow" the pattern, and try to see if there is a matching part of the graph. A more detailed description of this algorithm follows.

Recall that `pattern` is a list of tuples of the form `(label, neighbors)` where each list index refers to a distinct node.

1. Pick an index in the pattern that hasn't been assigned.
2. For every node in the graph:
    - If the label of the node matches the label of the index, assign the node to the index, else try next one.
    - For every neighbor of the current index in the pattern:
        - Repeat step 2 with the neighbor as the starting index and considering only the neighbors of the chosen node in the graph.
3. Repeat steps 1 and 2 until all nodes in the pattern have been assigned.

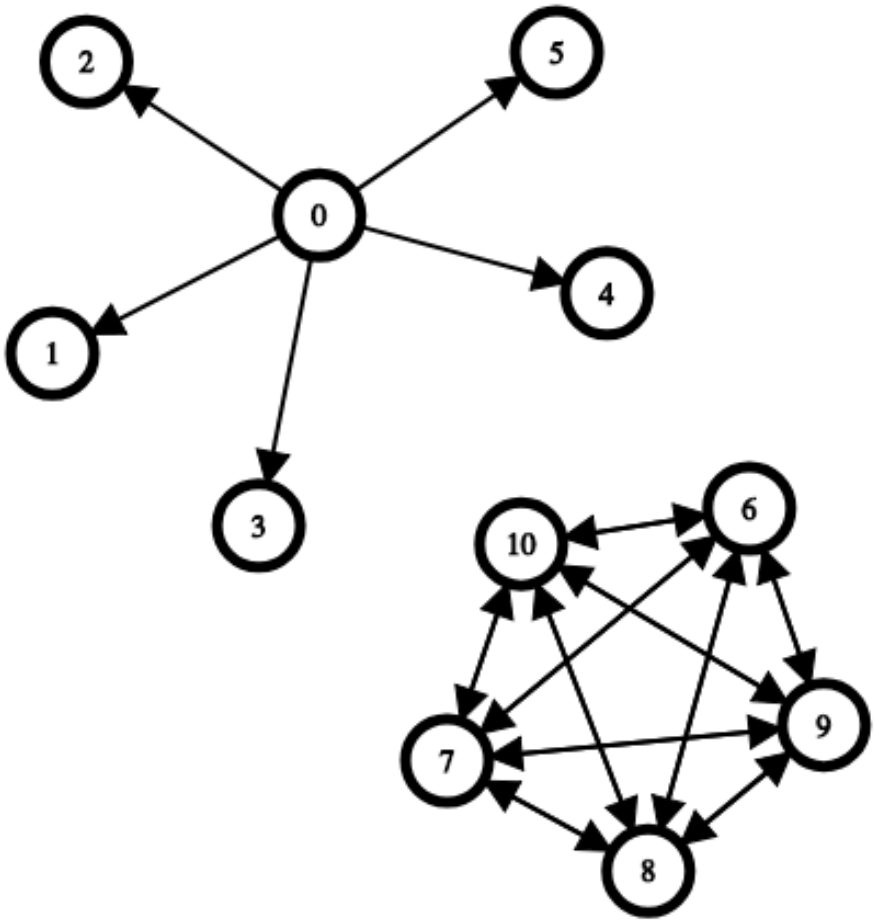
Keep in mind that you are looking for **all** possible assignments of nodes to the indices in the pattern, not just a single one. This means that you will need to figure out a way to keep track of these possible assignments as you are trying to cover the entire pattern with nodes from the graph.

Hint: You might find it useful to write a helper function to perform step 2. of the algorithm, since it is called recursively.

4) More Optimizations

Your new algorithm is much faster than the old one, but there is still room for improvement. Most of the queries Santa makes for the gift-delivery mission involve patterns that represent stars and cliques. An `n`-legged star is a structure that contains one central node that has edges towards `n` other nodes. `k`-cliques are subgraphs with `k` nodes in which any two nodes are connected by an undirected edge. Remember from Lab 6 that an undirected edge is represented as two directed edges in opposite directions.

Below you can find two examples of how such subgraphs would look: a 5-legged star and a 5-clique.

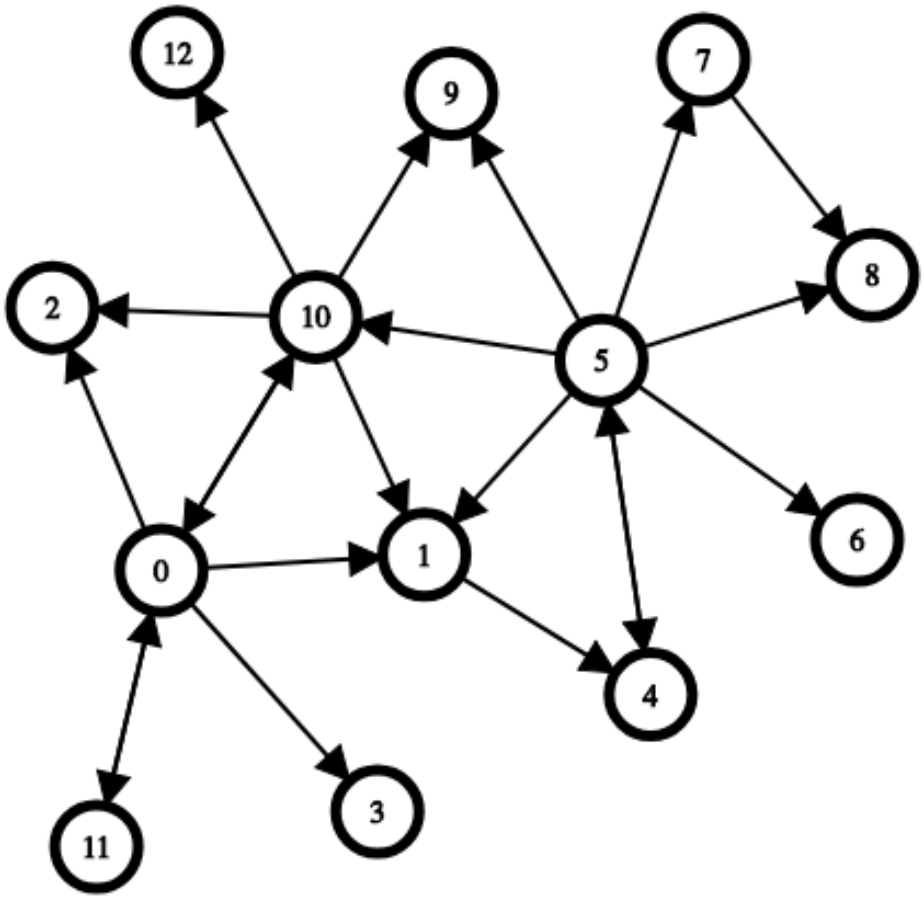


Santa is asking you to build on your current `FastGraph` implementation and add algorithms that would allow quicker queries for star and clique patterns. In order to support these optimizations, you will augment `FastGraph` with additional data structures that would make information relevant for the special star and clique queries quicker to access. In Lab 6, you created a `Graph` implementation that was more space-efficient at the cost of additional runtime, while now we will trade memory for time.

The improved data structure should be able to support general pattern queries the same way as before. The difference is that now patterns that resemble cliques and stars will be detected in advance, and you will apply a different, optimized version of `query` to them.

4.1) Stars

In order to optimize the graph search for star-like structures, you should think about how to store information about node degrees. The degree of a node is the number of total edges that are outgoing from that node to other nodes in the graph.[1] For example, the center node of the star above has degree 5. In order to better understand how this might be helpful in the context of our optimization problem, answer the questions below:



What is the degree of each node in the above graph? Enter your answer as a Python dictionary mapping node labels (integers) to the degree of the corresponding node.

{0:5, 1:1, 2:0, 3:0, 4:1, 5:7, 6:0, 7:1, 8:0, 9:0, 10:5, 11:1, 12:0

Submit

You have submitted this assignment 2 times.

This question is due on Monday April 22, 2019 at 04:00:00 PM.

Based on your answer to the previous question, what nodes from the above graph would be possible candidates for matching the central node of the 5-legged star pattern in the first example? Enter your answer as a Python set of integers in the box below.

set([0,10,5])

Submit

You have submitted this assignment 2 times.

This question is due on Monday April 22, 2019 at 04:00:00 PM.

Check Yourself:

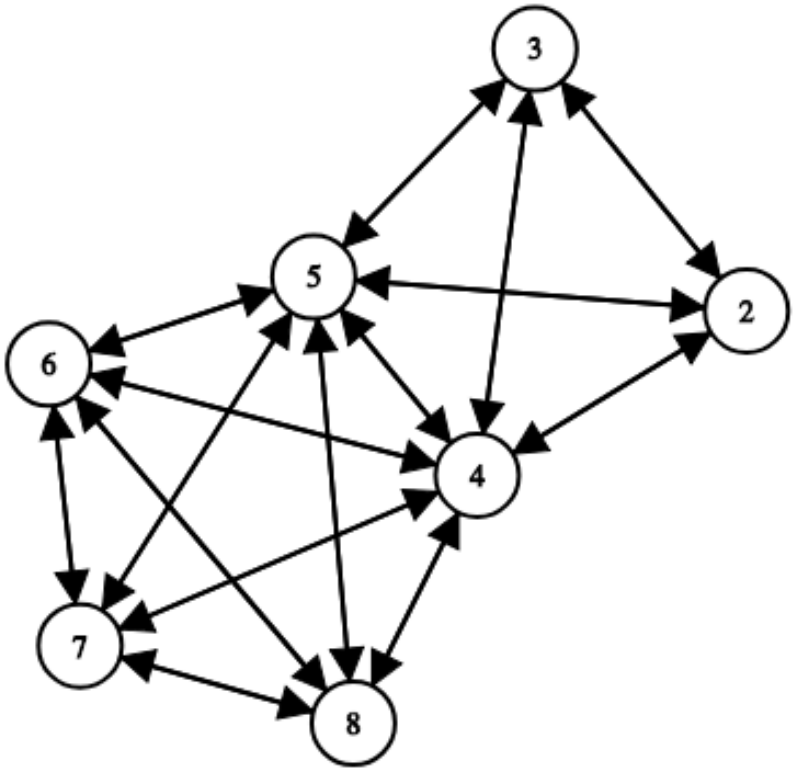
How do the node degrees change as one adds or removes nodes and edges from a graph? How can you use node degrees in order to make a query run faster on certain graphs?

After implementing support for faster star queries inside `FastGraph`, your code should be able to pass all test cases in the `Test_3_QueryStars` class from `test.py`.

4.2) Cliques

Currently, `query` tries to match each node in the graph for at least one position in the pattern, then follows each edge to make sure they exist in the graph as well. However, if the pattern is a sizable `k`-clique, it's likely that only a small fraction of the nodes in the graph will have the necessary connectivity to satisfy it (i.e. will be contained in cliques of size at least `k`). You could save a lot of work by storing sets of nodes that form cliques in the internal representation of `FastGraph`. This would allow the `query` method to only check a subset of the nodes when a clique pattern gets passed as argument. Note that nodes within a clique might still have different labels that need to be matched.

We suggest keeping track of either all the maximal cliques or just all cliques in which each node is contained while constructing the graph. A maximal clique is a clique whose size cannot be increased by including more nodes in the graph to it. In other words, the nodes in a maximal clique are not contained within any other, larger, clique. Note that a maximal clique need not be the maximum clique (the clique containing the largest number of nodes). To check your understanding, answer the concept questions below about the following graph, where node names are integers.



What are the sizes of the maximal cliques that node 5 is contained in? Enter your answer as a Python set of integers.

set([5,4])

Submit

You have submitted this assignment 1 time.

This question is due on Monday April 22, 2019 at 04:00:00 PM.

Let's add a new node to the graph with *undirected* edges to nodes 4 and 5. What are the sizes of maximal cliques 5 is contained in now? Enter your answer as a Python set of integers.

set([5,4,3])

Submit

You have submitted this assignment 3 times.

This question is due on Monday April 22, 2019 at 04:00:00 PM.

If we remove the edge(s) between nodes 4 and 5, what are the sizes of all the maximal cliques present in the graph? Enter your answer as a Python set of integers.

set([3,4])

Submit

You have submitted this assignment 2 times.

This question is due on Monday April 22, 2019 at 04:00:00 PM.

Which approach would be more *space* efficient: maintaining all *maximal* cliques or maintaining all possible cliques that each node is contained in? Since our focus in this lab is on *time* efficiency and because these two strategies would make the `query` method similarly fast, it is up to you which one to use. However, note that keeping track of maximal cliques might introduce additional complications that you need to account for. For example, some maximal cliques might become non maximal after adding edges to the graph and some non maximal cliques might become maximal after removing edges/nodes from the graph.

Regardless of which strategy you use, the number and dimensions of the cliques can still change significantly in the process of building a graph through the methods defined on the `Graph` interface. It is up to you how to store these cliques as well as the relationship between a node and the cliques that contain it, but you should make sure that the methods which mutate the graph by adding or removing nodes/edges properly update the internal representation of `FastGraph`. If you need help, you can take a look at the following **hints** on how certain graph operations can affect the clique dynamics.

You might notice that the `add_edge`, `remove_edge`, `add_node` and `remove_node` methods actually become slower than before implementing the clique optimization, but this is nothing to worry about! Santa rarely changes the maps and usually only uses your data structure for multiple repeated queries on cliques and stars. This means that if the `query` method is fast enough on these two kinds of inputs, the time saved when performing queries offsets the time taken to build the graph.

After implementing this feature, your code should pass all test cases in `test.py`.

5) Code Submission

Download Your Last Submission
Click to View Your Last Submission
Select File No file selected
Submit
You have submitted this assignment 7 times.
This question is due on Monday April 22, 2019 at 04:00:00 PM.

6) Checkoff

Once you are finished with the code, please come to a tutorial, lab session, or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular, be prepared to discuss:

- your `query` implementation in `FastGraph`
- description of your optimization for stars
- description of your optimization for cliques

6.1) Grade

Grading:

- Concept questions (0.5 points): 0.5
- Tests (1.5 points): 1.5
- Checkoff (2 points): 2

Total: 4 Points (of 4)

Comments from Grader:
good job!

Footnotes

[1] Strictly speaking, this is the definition of out-degree. There is also in-degree, which is the number of edges incident to the node.