Lab 6: 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: `lab6.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

It is Christmas Eve, and Santa Claus needs to deliver gifts to the children of Algoville. The night is dark, but worry not! Santa has an electronic version of the map of the city. However, this map is too large for Santa to view at once. The data stored in the map can be used via specially formatted queries. In this lab, we will ask you to implement an idealized version of this interface in two different ways. In a subsequent lab, you will explore how these representations can be optimized further.

2.1) The `Graph` Interface

We will represent our city as a mutable graph. In this graph, the nodes are the city's homes, and the edges between them are the streets connecting them. Each node is guaranteed to have a `name`, which is a unique identifier for that node, and might have a `label` (labels are not mandatory and if present, they don't need to be unique). The reason node labels can be useful is that they allow filtering only one category of nodes (buildings, intersections, etc). Since streets can be one-way

(and Santa respects traffic rules), our graph will have directed edges (undirected edges can be replaced with two directed edges).

To interact with this data structure, we provide the `Graph` interface, which you can see in `graph.py` in the code distribution for the lab. Python does not have a separate "interface" concept that is distinct from classes, so we have provided you the interface as a class.

Since `Graph` is an interface, its methods should remain unimplemented. Remember that the role of an interface is to provide a consistent way of interacting with the data structure and hide the specifics of the implementation. This way, changes in the different implementations do not affect the methods a user (Santa) will use to modify the map or perform queries on it.

There are four methods available to modify the graph:

- `add_node(self, name, label='')`: add a new node to the graph with name `name` and label `label`. The default label is the empty string. The label `'*'` should not be used, since it has a special meaning (see the description of `query`). If a node with the same name already exists, this method should raise a `ValueError`.
- `remove_node(self, name)`: remove a node previously added via `add_node`. If a node with the given name doesn't exist, this method should raise a `LookupError`.
- `add_edge(self, start, end)`: add a directed edge from node with name `start` to node with name `end`. If either of these nodes doesn't exist, this method should raise a `LookupError`. If the edge already exists, it should raise `ValueError`.
- `remove_edge(self, start, end)`: remove the edge from node `start` to node `end` (but not in the opposite direction). If either `start`, `end`, or the edge between them doesn't exist, this method should raise `LookupError`.
- `query(self, pattern)`: returns a list of lists representing all combinations of **distinct** nodes that match the given pattern.

The information stored in a graph can be accessed via the method `query` above. The only parameter of this method is `pattern`, which represents a graph (or a set of different graphs) and is used to match parts of our graph. `pattern` is a list of tuples, where each tuple represents a node. Each tuple has two elements: the `label` of the node in question and a list of its neighbors. The labels in the tuples of the pattern can be any string. However, a single asterisk `'*'` should be treated specially and is understood to match any label. The "neighbors" are integers which represent indices into `pattern`.

For example, the pattern `[('fancy house', [1]), ('shopping mall', [])]` would match any two nodes where the first one has label `'fancy house'` and is connected by a directed edge to a second node with label `'shopping mall'`. These nodes can still have edges to other nodes within the graphs (including an edge *from* the node labeled `'shopping mall'` *to* the node labeled `'fancy house'`).

The return value of `query` is a list of lists, which represents the matches. Each of these lists should have the same length as `pattern` and should contain the **names** of the nodes matching the queried nodes in `pattern` in the same order. For example, in our graph, there is a node labeled `'fancy house'` called `'North Palace'` and two nodes labeled `'shopping mall'`: `'South Center'` and `'Grand Plaza'`. There are edges from `'North Palace'` to both `'South Center'` and to `'Grand Plaza'`. Therefore, `query` would return the list `[['North Palace', 'South Center'], ['North Palace', 'Grand Plaza']]`. The order of the results in the top-level list doesn't matter, but they should be unique (so repetitions are not allowed).

What query could be used to match **any** single node in the graph?

`[('*', [])]`

Submit

You have submitted this assignment 6 times.

This question is due on Friday April 05, 2019 at 04:00:00 PM.

What query could be used to match **any** two nodes with a single directed edge between them?

[('*', [1]),('*', [])]

Submit

You have submitted this assignment 1 time.

This question is due on Friday April 05, 2019 at 04:00:00 PM.

What query could be used to match **any** two nodes with edges between them in both directions?

[('*', [1]),('*', [0])]

Submit

You have submitted this assignment 3 times.

This question is due on Friday April 05, 2019 at 04:00:00 PM.

3) Implementations of the Graph Interface

3.1) Factory Methods

Before we start implementing `Graph`, we need to think about how to create instances of this interface (and its implementations). Of course, given a real graph, we could manually add all the nodes and edges in it to an empty `Graph` object, but it would be much nicer to have another class do this for us!

We are concerned about two particular ways to represent graphs using built-in Python objects. One of them is an adjacency list, which is a list of lists. Each inner list represents a node as a list of its neighbors' names. The neighbors of a node are all the nodes towards which that node has a directed edge. The names of the nodes are simply their indices inside the top-level list. Therefore, the neighbors of a node are the nodes found at the indices from its neighbor list.

For example, a cycle containing 4 nodes would have the following adjacency list: `[[1], [2], [3], [0]]`.

The other representation is an adjacency dictionary, where each key is a node name (need not be an integer as in the adjacency list) and the value associated with the key is a list of its neighbors (so a list of node names).

Implement the class `GraphFactory` in `lab.py`. The `__init__` constructor takes the `graph_class` argument, which is any class implementing the `Graph` interface (classes can be passed around in Python, just like functions and other objects). This instance of `GraphFactory` should create instances of this specific class. Note that `graph_class` is guaranteed to be a subclass of the `Graph` interface. Therefore, all of the methods defined for `Graph` can also be used on objects of type `graph_class`.

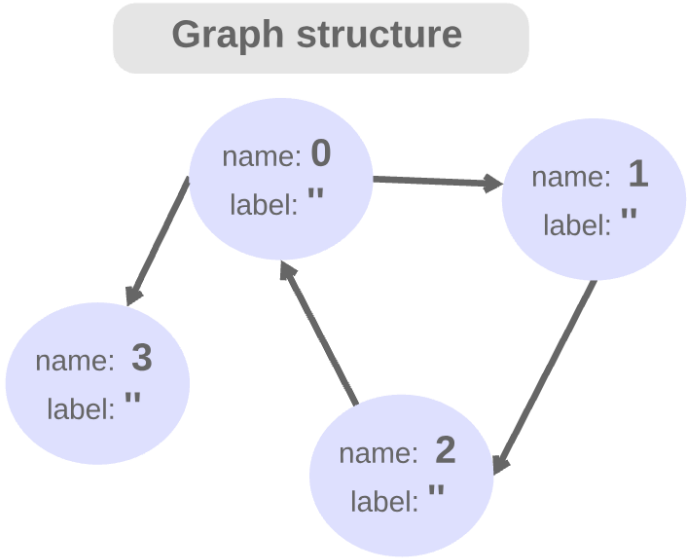
The only methods of `GraphFactory` are `from_list` and `from_dict`, which create instances of `graph_class`. The optional argument `labels` is a dictionary mapping node names to node labels (if it's `None`, you should not explicitly set node labels).

To better understand how the GraphFactory class would be used, imagine you have access to classes `Graph1`, `Graph2` and `Graph3`, all inheriting from `Graph`. If you would like to obtain instances of `Graph1`, then you would use the following piece of code:

```
>>> graph_factory = GraphFactory(Graph1)
>>> graph_instance = graph_factory.from_list(adj_list)
```

Your GraphFactory should be able to instantiate any subtype of Graph, not only the two classes you will implement in this lab. If your implementation is based on methods or attributes other than the ones provided in the Graph interface, it might fail some of the test cases.

```
adj_list = [[1, 3], [2], [0], []]
factory = GraphFactory(SimpleGraph)
graph = factory.from_list(adj_list)
```



`adj_list`

is a list of lists. The index i of each inner list contains the names of node i's neighbors.

In the example on the left, from the first inner list of `adj_list`, `adj_list[0]`, we see that the node with name 0 has a directed edge going from it to nodes with names 1 and 3.

From `adj_list[3]`, we see that the node with name 3 does not have any directed edges going from itself to other nodes.

Note that if instead we had the following adjacency list:
`adj_list = [[1,3], [2], [0]]`
and then proceeded as follows:
`graph=factory.from_list(adj_list)`

we would get a LookupError when adding an edge from node 0 to node 3, since `adj_list` does not contain anything at index 3, indicating that a node with name 3 does not exist in the graph.

3.2) Simple Representation

We will start with a rather simple implementation, and will make it progressively better. We would like you to implement a class called `SimpleGraph`, which should be a subclass of the interface `Graph`, implementing all of its methods (you should not modify the methods in `Graph` itself). For the internal graph representation of this simple version, we suggest using a simple adjacency dictionary, where each node is a key, and the values represent their neighbors.

Hint: writing the query method efficiently is hard. However, for the purposes of this lab, exhaustively enumerating all node sequences that have the same length as `pattern` and match the labels of the pattern should be fast enough. Note that there is no point in generating a permutation if the labels of its nodes do not match the pattern labels, and doing so might not be efficient enough to pass some of the test cases.

3.3) Reducing Redundancy

Now that we have a simple version of our implementation, we can think about how to make it better. In a city, many central places can be reached via streets from most other large intersections. This means that it is possible that we will end up with many nodes in our graph having the same exact set of neighbors.

One special such example is a complete graph, in which each node has an edge to any other node in the graph, including to itself. If you were to compute the set of neighbors of each node in this graph, how many distinct sets would you obtain?

Can you think about an optimization exploiting this redundancy? Create `CompactGraph`, a second implementation of our `Graph` interface, which should be more space-efficient than `SimpleGraph`. Note that some of our tests will fail on out-of-memory errors if your implementation doesn't exploit redundancy well enough.

Hint: Instead of the adjacency dictionary used in `SimpleGraph`, choose another internal representation for `CompactGraph` that takes less space when the number of distinct neighbor sets in a graph is small.

4) Delivery Team Allocation

Santa is impressed by our query tool so far but would need some more help to set up its first application in the gift-delivery mission: distributing delivery tasks. Instead of delivering all the gifts alone, Santa hired elves in order to speed up the process. To help with this, we need to develop an algorithm that allocates a certain number of delivery teams for those gifts that are in high demand in certain areas of the city.

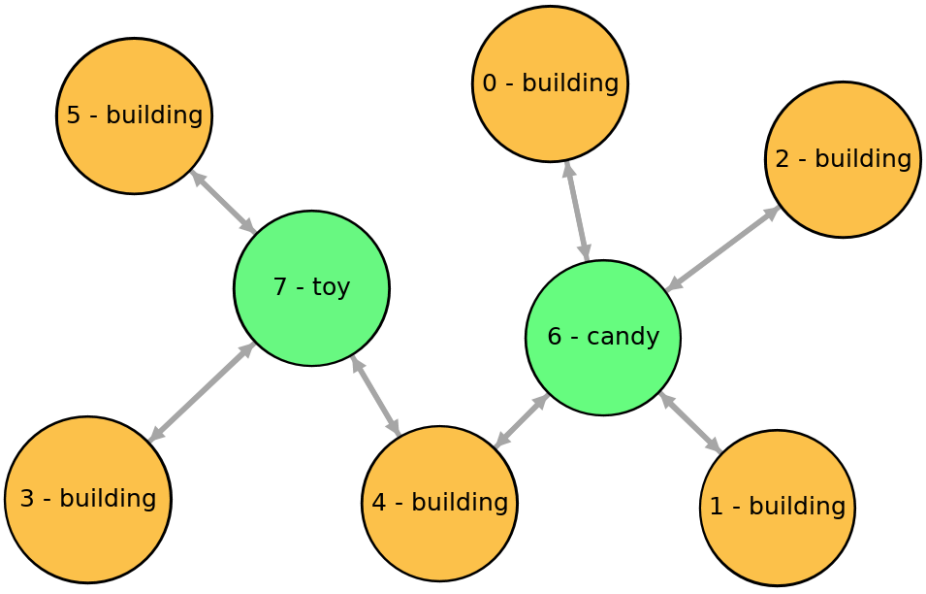
Fortunately, the `Graph` interface that we have been developing can be used for this purpose as well! Santa has a map depicting each building in the city and the gifts that need to be delivered there. More specifically, each building is a node with label `'building'`, and each gift is a node with a unique label describing it, like `'puppy'`, or `'candy'`. There is an undirected edge between a building node and a gift node only if Santa plans to deliver that gift to the corresponding building. Remember that an undirected edge between two nodes is equivalent to two opposite, directed edges.

Unlike Santa, the elves use the subway to travel and can only carry a single type of gift each. The minimum wage in Algoville is fairly high, so Santa would only send a team of elves to deliver gifts if there is at least a group of *k* (a number determined by Santa) buildings or more that are close to the same subway station and require a certain type of gift. In other words, Santa will need one team for each set of at least *k* buildings that are connected to a gift and that cluster around the same station, so that the elves don't have to walk for too long.

We need to come up with an algorithm that helps Santa figure out the minimum number of delivery teams needed to send for each type of gift in the graph, using the graph and number *k* described above, as well as a dictionary mapping the name of each building node to the name of the closest subway station and a set of gift labels. Our function should return a dictionary in which keys are labels of the gift nodes and values are the numbers of delivery teams required to deliver the corresponding gifts.

For example, for `k = 3` and the subway-station dictionary below, the following graph would require one team for the candy gift and one team for the toy gift. All the other nodes that are not part of big enough clusters, and so are not reached by delivery teams, are exceptions, and Santa will make sure to deliver to those buildings separately.

```
>>> stations = {0: "Old Square", 1: "Old Square", 2: "Old Square", 3: "Town Hall", 4: "Town Hall", 5: "Town Hall"}
>>> k = 3
```



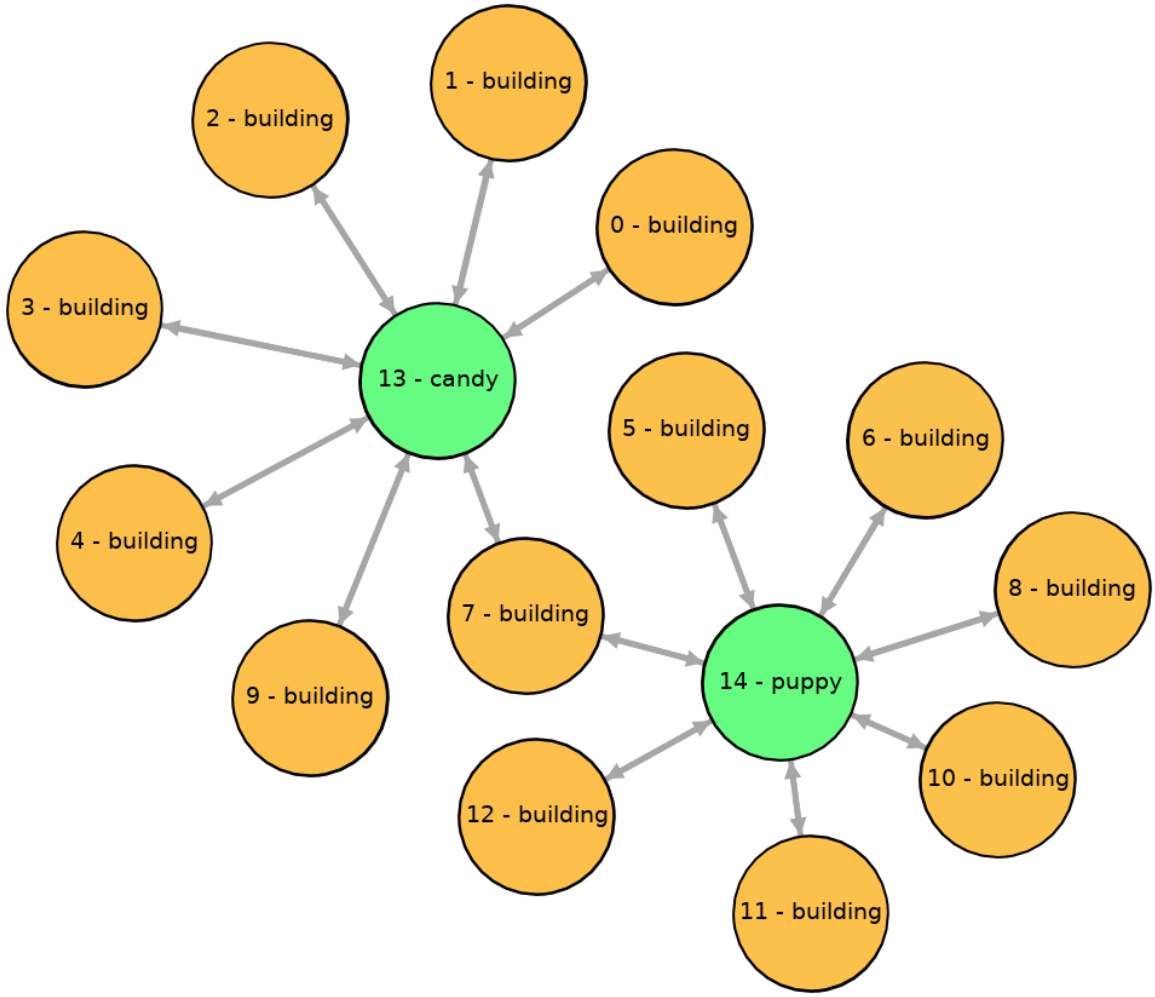
Note that the graph has been color-coded to show the buildings in orange-yellow and the gifts in green. The numbers on the nodes are node names, while the text are the labels.

The resulting team allocation is:

```
allocation = {"candy": 1, "toy": 1}
```

Now let's look at a larger graph, with the following station dictionary:

```
>>> stations = {0: "Old Square", 1: "Old Square", 2: "Old Square", 3: "Old Square", 4: "Old Square", 5: "Town Hall", 6:
"Town Hall", 7: "Town Hall", 8: "Town Hall", 9: "South Station", 10: "Ice Rink", 11: "Ice Rink", 12: "Ice Rink"}
```



What is the smallest *k* for which Santa cannot send any elf teams and would have to deliver **all** gifts alone?

6

Submit

You have submitted this assignment 1 time.

This question is due on Friday April 05, 2019 at 04:00:00 PM.

If Santa sets `k = 3`, how many delivery teams will be sent for each type of gift? Enter your answer as a Python dictionary in the box below

{'candy':1, 'puppy':2}

Submit

You have submitted this assignment 1 time.

This question is due on Friday April 05, 2019 at 04:00:00 PM.

Now suppose `k = 4`, how will the team allocation look in this case? Enter your answer as a Python dictionary in the box below

{'candy':1, 'puppy':1}

Submit

You have submitted this assignment 1 time.

This question is due on Friday April 05, 2019 at 04:00:00 PM.

Now that you have a better idea of what Santa wants the new feature to do, it is time to think through the algorithm. The first step towards solving this allocation problem is to figure out how to incorporate information about the closest station of each building into our graph. If you need help with this part, we provide **a few hints**.

4.1) Finding the number of elf teams Santa needs

Given a graph of building and gift nodes and *k*, the minimum size a cluster of buildings needs to have for Santa to send a delivery team there, as well as a dictionary mapping node names to subway stations and sets of gift labels, implement the `allocate_teams` function inside `lab.py`.

Note that you are not allowed to add any methods or attributes to the `Graph` interface in order to retrieve node or edge information about the graph (like finding the names of all nodes in the graph). Your function should work with any underlying implementation of the interface.

Hint: You should be able to obtain a list of all nodes in a graph by passing the right pattern to the `query` method. How does that pattern look?

5) Using the UI

We have also provided a visualization website which loads your code into a small server (`server.py`) and visualizes your results. To use the visualization, run `server.py` and navigate to localhost:8000 in your web browser. You will need to restart `server.py` in order to reload your code if you make changes.

You will be able to see the nodes as small circles with the name and label written on them (in this order), separated by a colon. The edges between the nodes are symbolized by arrows (a double-headed arrow means that there are edges in both directions between the nodes).

Above the graph, you will be able to choose a `Graph` implementation to use from your `lab.py` and a query to run. The matched subgraphs will be highlighted (nodes in green, edges in red).

6) Code Submission

Download Your Last Submission
Click to View Your Last Submission
Select File No file selected
Submit
You have submitted this assignment 3 times.
This question is due on Friday April 05, 2019 at 04:00:00 PM.

7) 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 and test cases 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 implementation of the factory methods in `GraphFactory`
- your implementation of `SimpleGraph`
- description of what optimizations you added to `CompactGraph` and why those work
- your implementation of `CompactGraph`
- description of the algorithm you used for allocate_teams
- your implementation of allocate_teams

7.1) Grade

Grading:

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

Total: 4 Points (of 4)