CS 231
Naomi Nishimura

Module `graphs.py`

1 Introduction

Just like the Python list and the Python dictionary provide ways of storing, accessing, and modifying data, a *graph* can be viewed as a way of storing, accessing, and modifying data. Because Python does not have built-in support for graphs, I have supplied a module for use in the course.

This document provides background on the types of graphs supported by the module as well as specifics of the functions provided. The module is not intended to cover all kinds of graphs nor to provide all possible functions. Details of the implementations of the functions are not discussed here; to learn more about such implementations, consider taking CS 234.

At times you may be writing pseudocode that uses graph operations. To use a graph function, simply translate from dot notation, put the name in all capital letters, and capitalize the first letter in each variable name. For example, instead of `graph.edge_label(one, two)`, write EDGE_LABEL*(Graph, One, Two)*.

2 Graph basics

A graph consists of a set of *vertices* and a set of *edges*, where each edge is a pair of vertices, called the *endpoints* of the edge. We allow at most one edge between a pair of vertices, so the edges must all be distinct. You can think of edges as representing connections between vertices.

If two vertices are endpoints of an edge, they are *neighbours*. The set of neighbours of a particular vertex is its *neighbourhood*. The number of neighbours of a vertex is the *degree* of the vertex, denoted $\deg(v)$ for vertex v. We will require that each edge have two different vertices as its endpoints, so a vertex will never be its own neighbour.

Two vertices are *adjacent* if they are endpoints of an edge, and two edges are *incident* if they share an endpoint. The term *incident* is also used to describe the relationship between an edge and its endpoints.

A sequence of vertices forms a *path* if there is an edge between each consecutive pair of vertices in the sequence, and a *cycle* if there is also an edge between the first and last vertices in the sequence.

At times we might talk about a graph representing another graph. In that case, we may use the term *node* as a synonym for vertex. Typically we will refer to vertices in the original graph and nodes in the representation graph.

Two special types of graphs that we might consider are graphs that are *connected* (there exists a path between each pair of vertices) and that are *complete* (there exists an edge between each pair of vertices).

3 Module `graphs.py`

3.1 Objects

The graph module makes use of three different types of objects: Vertex, Edge, and Graph. In the graphs used in the course, each Vertex object has an ID, a label, a weight, and a colour, where the weight is an integer and all other attributes are strings; the default values of the label, weight, and colour are `"none"`, `0`, and `"white"`. The ID of each Vertex must be unique. Each Edge object consists of the IDs of two Vertex objects, a label, a weight, and a colour. Notice that edges do not have IDs; each edge can be uniquely identified by the IDs of its two endpoints.

3.2 Creating a graph from a file

The module `graphs.py` contains the function `make_graph`, which consumes a string (the name of a file) and produces an object of type Graph.

A file containing graph data should contain the following information, in this order:

- the number of vertices in the graph (on one line),
- ID, label, weight, and colour of a vertex (four values per line, where weight is an integer and the other values are strings), and
- IDs of both endpoints, label, weight, and colour of an edge (five values per line, where weight is an integer and other values are strings).

For a graph with n vertices and m edges, there will be a total of $1 + n + m$ lines in the file. Please note that when creating a graph from a file, you need to specify labels, weights, and colours, even if you wish to use the defaults.

3.3 Creating a text file for a new graph

The module `graphs.py` contains the function `make_graph_text_file`, which consumes three strings (`vertex_string`, `edge_string`, and `file_name`) and produces a text file that can be used to create a graph. You may find this function useful when creating graphs to use when testing your code. This function was generously provided by Adam Hunter, who took the course in Spring 2018 and wished to make life easier for future students.

The three strings should contain the following information:

- `vertex_string` is a string of distinct characters (no spaces)
- `edge_string` is a string of pairs of characters from `vertex_string` (no spaces)
- `file_name` is a suffix for the name of the file

The file will represent a graph in which there are vertices with IDs for each character in `vertex_string` and edges between vertices with specified IDs, as represented by pairs of characters in `edge_string`. Each vertex and each edge will have the label `"none"`, the weight `0`, and the colour `"white"`. The name of the file will be `"testgraph"` concatenated with `file_name`.

For example, `make_graph_text_file("abcd", "abbccddbcada", "3-1")` will create a graph with the name `testgraph3-1.txt` that contains the following data:

```
4
a none 0 white
b none 0 white
c none 0 white
d none 0 white
a b none 0 white
b c none 0 white
c d none 0 white
d b none 0 white
c a none 0 white
d a none 0 white
```

3.4 Methods

The table below lists the methods that can be used on objects of the class Graph. Pay close attention to the types consumed and produced by each method; sometimes you will be handling objects and sometimes you will be handling string IDs. The file `graphuse.py` gives an example of the methods being used.

You can use `print` to print a single Vertex, Edge, or Graph. If you wish to print a list of edges, create a for loop and print each edge in the list.

Due to the way that a graph is implemented, a list produced by a method may not have the items appear in a predictable order. Please see information on the module `equiv.py` for functions to use in such situations.

You need to ensure that IDs are distinct for all vertices; the code will not check for you. Similarly, you should ensure that when you supply the endpoints of an Edge, such an Edge exists. The order in which the endpoints are supplied is not important. That is, the Edge with endpoints with IDs `"a"` and `"b"` is identical to the Edge with endpoints with IDs `"b"` and `"a"`; the two orders of endpoints are two ways of referring to the same Edge.

Because the module is designed to allow you to implement code with graphs without considering the details of how the graph is implemented, the worst-case costs listed in the table are not intended to reflect the actual costs of this particular implementation. When writing an algorithm for graphs, one often chooses among various options with differing costs for operations. The costs listed here are not the best possible, but a reasonable choice that you can use for analysis. Here we use n to denote the number of vertices in `graph`, m to denote the number of edges in `graph`, and d to denote the degree of the vertex with ID `one`; note that $d \in O(n)$. In addition, you can assume that access to any of the attributes can be accomplished in constant time.

Method	What it does	Cost
`Graph()`	creates a new empty graph	$\Theta(1)$
`repr(graph)`	produces a string of information about vertices and edges in `graph`	$\Theta(nm)$
`graph.vertices()`	produces a list of IDs of vertices in `graph`	$\Theta(n)$
`graph.edges()`	produces a list of Edge objects in `graph`	$\Theta(m)$
`graph.neighbours(one)`	produces a list of IDs of neighbours of the vertex with ID `one`	$\Theta(d)$
`graph.are_adjacent(one, two)`	produces True if vertices with IDs `one` and `two` are adjacent and False otherwise	$\Theta(d)$
`graph.vertex_label(one)`	produces the label of the vertex with ID `one`	$\Theta(1)$
`graph.vertex_weight(one)`	produces the weight of the vertex with ID `one`	$\Theta(1)$
`graph.vertex_colour(one)`	produces the colour of the vertex with ID `one`	$\Theta(1)$
`graph.edge_label(one, two)`	produces the label of the edge between vertices with IDs `one` and `two`	$\Theta(d)$
`graph.edge_weight(one, two)`	produces the weight of the edge between vertices with IDs `one` and `two`	$\Theta(d)$
`graph.edge_colour(one, two)`	produces the colour of the edge between vertices with IDs `one` and `two`	$\Theta(d)$
`graph.add_vertex(one)`	adds a new vertex with ID `one`	$\Theta(1)$
`graph.del_vertex(one)`	deletes the vertex with ID `one` and all edges with the vertex as an endpoint	$\Theta(m)$
`graph.add_edge(one, two)`	adds a new edge between vertices with IDs `one` and `two`	$\Theta(1)$
`graph.del_edge(one, two)`	deletes the edge between vertices with IDs `one` and `two`	$\Theta(m)$
`graph.set_vertex_label(one, new)`	updates the label of the vertex with ID `one` to `new`	$\Theta(1)$
`graph.set_vertex_weight(one, new)`	updates the weight of the vertex with ID `one` to `new`	$\Theta(1)$
`graph.set_vertex_colour(one, new)`	updates the colour of the vertex with ID `one` to `new`	$\Theta(1)$
`graph.set_edge_label(one, two, new)`	updates the label of the edge between vertices with IDs `one` and `two` to `new`	$\Theta(d)$
`graph.set_edge_weight(one, two, new)`	updates the weight of the edge between vertices with IDs `one` and `two` to `new`	$\Theta(d)$
`graph.set_edge_colour(one, two, new)`	updates the colour of the edge between vertices with IDs `one` and `two` to `new`	$\Theta(d)$
`graph_a` == `graph_b`	produces `True` if `graph_a` and `graph_b` have vertices with the same IDs and the same edges (but labels, weights, and colours can differ)	see note below

For your convenience, the module also allows you to check for equality of graphs (use this

only for tests, please), where vertex IDs and edges much match but labels, weights, and colours can differ.
Note: More precise analysis may be possible by using the observation that the sum of the degrees of the vertices in a graph is equal to twice the number of edges. (To see why this is true, observe that the degree of a vertex is the number of neighbours it has, which is equal to the number of edges for which it is an endpoint. Since each edge has two endpoints, summing over the degrees counts each edge twice.) For assignment and exam questions in this course, unless stated otherwise, you may use the less precise bound obtained by using the fact that $d \in O(n)$.

4 Using the module to write code

The module uses the module `equiv.py`, so make sure that you have downloaded the module before using `graphs.py`.

4.1 Writing your own graph functions

You can also write your own graph functions by directly accessing the attributes in the classes `Vertex` and `Edge`. In calculating costs, you can assume that access to any of the attributes can be accomplished in constant time.

A `Vertex` object has the following attributes:

- `id` (a string)
- `label` (a string)
- `weight` (an integer)
- `colour` (a string)

An `Edge` object has the following attributes:

- `vertex_u` (a Vertex object)
- `vertex_v` (a Vertex object)
- `label` (a string)
- `weight` (an integer)
- `colour` (a string)

It is not recommended that you directly access the attributes of the class Graph, as determining the costs requires knowledge extraneous to this course.

4.2 Copying graphs

If you wish to make a copy of a graph, import the `copy` module and use `copy.deepcopy`.

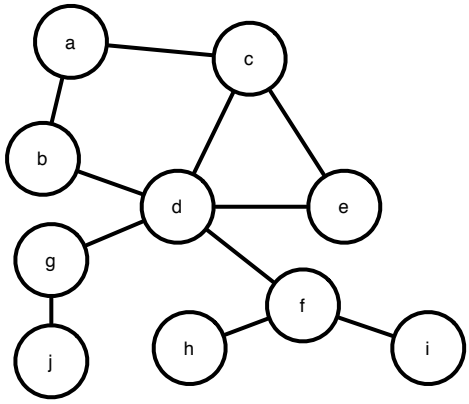

Figure 1: Sample graph 1

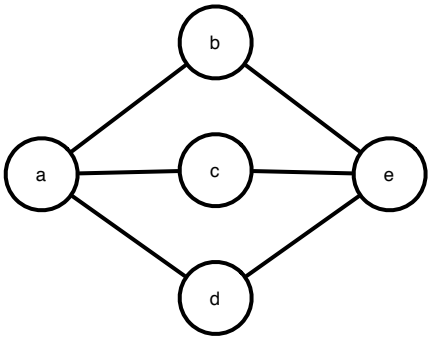

Figure 2: Sample graph 2

5 Sample graphs

Sample graphs have been provided for you in the files `samplegraph1.txt`, `samplegraph2.txt`, `samplegraph3.txt`, `samplegraph4.txt`, and `samplegraph5.txt`. For your convenience, they have been illustrated here. Code that you write for assignments should work for any graph, not just the samples provided. The vertices are labeled with the vertex IDs. The labels on the edges of the last two sample graphs are the weights of the edges.

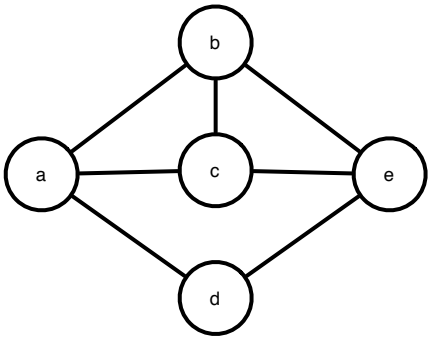

Figure 3: Sample graph 3

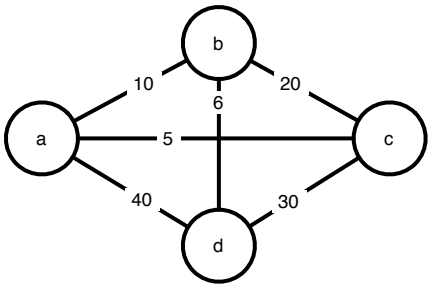

Figure 4: Sample graph 4

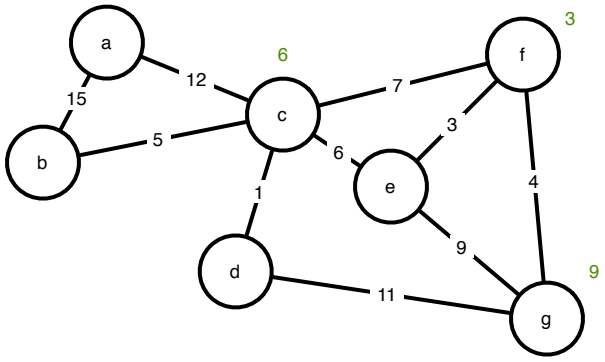

Figure 5: Sample graph 5