Lab 5: Graphs

October 30, 2016

Contents

1 Objectives

By the end of this lab, you will:

- understand two ways to represent graphs programmatically.
- learn how to define and create graphs in Pyret.
- understand and implement the Bellman-Ford algorithm.

Note: Portions of this lab are adapted from *Programming and Programming Languages (2016): Chapter 15, Graphs.*

2 Graphs in Code

How do we represent a graph? In our day to day lives, the answer is usually through a visual, maps being the most common example. Translating these visual graphs into code, however, can be more complicated than one might think. While we tend to represent graphs visually, there are numerous ways to represent a graph on a computer. Which of these many representations we decide to use depends on a number of factors:

- The structure of the graph, and in particular, its density – a measure of how many edges a graph has compared to the maximum amount it could have.
- The representation in which the data are provided by external sources. Sometimes it may be easier to simply adapt to their representation; in particular, in some cases there may not even be a choice.

- The features provided by the programming language, which make some representations much harder to use than others.

No matter our choice of representation, there are several things that we need to be able to work with graphs efficiently:

- A way to construct graphs.
- A way to identify (i.e., tell apart) vertices in a graph.
- Given a way to identify vertices, a way to get a vertex's neighbors in the graph.

In this lab, we'll consider two possible representations of graphs: **links by name**, and as a **list of edges**.

First, let's talk about links by name. We will assume that every vertex has a unique name (such a name, when used to look up information in a repository of data, is sometimes called a key). A vertex is then a key, and a list of keys that refer to other vertices:

```
data Vertex:
  | vertex(key :: String, adj :: List<String>)
end

type Graph = List<Vertex>
```

(Here we're assuming our keys are strings.)

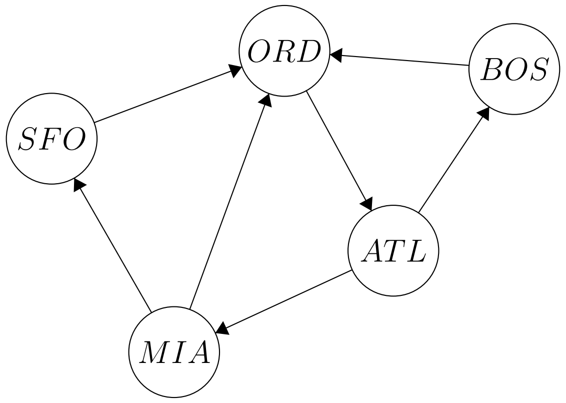
This representation gives priority to vertices, making edges simply a part of the information in a vertex. We could, instead, use a representation that makes edges primary, and vertices simply be the entities that lie at their ends:

```
data Edge:
  | edge(src :: String, dest :: String)
end

type Graph = List<Edge>
```

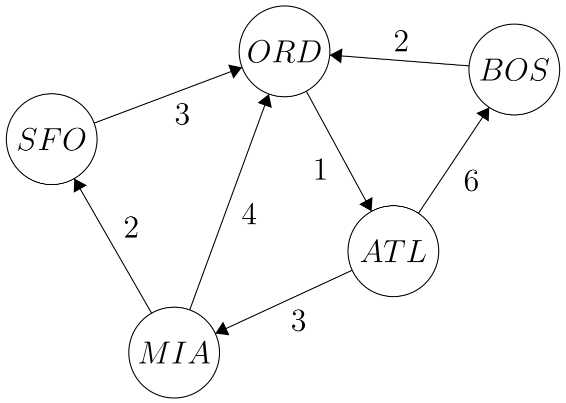
Task: The following directed graph models flight paths between cities, where each vertex is a city and each edge is a flight path. In Pyret, write two models for this graph using the two graphical representations just disccused. (**Note:** Keep in mind that the flights in this graph are one way! Make sure that your data model reflects this.)

*** Before continuing, call over a TA to check your work so far.**



3 Adding Some Weights

In many practical applications of graphs, there is some quantity associated with each edge. For example, in our flight graph, we might associate the distance of each flight. In a social network, the edge might include the number of messages sent between two people. In general, we refer to this quantity as the *weight* of an edge. If we add weights to our previous graph, we might obtain:



Task: In Pyret, try modifying the two graph representations discussed earlier so that they include a notion of weight. For which of the representations is this easier?

*** Before continuing, call over a TA to check your work so far**

4 Finding the Shortest Distance

Having added weight, we might then ask, what is the lowest-weight path between two vertices in a graph? In our flight example, this would represent the shortest sequence of flights (measured by distance) to get from your source to your destination. Numerous algorithms exist to compute this, such as Dijkstra's Algorithm, A*, and what we'll implement today: the Bellman-Ford algorithm.

Bellman-Ford, given a starting vertex V, will produce the shortest distances to all other vertices in the graph in an iterative fashion. It does this in the following way:

- Assign each vertex in the graph a cumulative weight of infinity, measured from V, with the exception of V itself, which has a cumulative weight of 0.
- Consider each edge, E, in the graph. If the cumulative weight to `E.src`, plus `E.weight`, is less than the cumulative weight to `E.dest`, then we have identified a shorter path to the destination vertex, so we set the cumulative weight of `E.dest` to `E.src + E.weight`.
- Run the previous step a total of $N-1$ times, where N is the number of vertices in the graph. We do this to ensure that we have identified the shortest path to every vertex in the graph. You can prove that the shortest path will be discovered within $N-1$ iterations, but that is beyond the scope of this lab.

Our task for the remainder of the lab is to implement this algorithm in Pyret. We've provided you with a stencil, which outlines the necessary functions and data types. Here's a brief overview:

To attach each vertex with a cumulative weight from our starting vertex, we've provided you with the following data structure:

```
data C-Weight:
  | c-weight(vtx :: String, w :: Number)
end
```

We will use the Option type to indicate whether we have computed the cumulative weight to a vertex or not, i.e. a cumulative weight of none indicates that we have not visited a vertex yet. These are the functions that we suggest you implement:

- `get-weight(vtx :: String, weights :: List<Weight>) -> Option<Number>`

  In this function, you will need to determine if the vertex is in the list. If it is not in the list, return none, otherwise, return some with that distance.
- `set-weight(vtx :: String, w :: Number, weights :: List<Weight>) -> List<Weight>`

  In this function, you will need to determine if the vertex is in the list. If it is not in the list, add it with the specified weight, otherwise, replace the current weight with the new weight.
- `get-vertices(graph :: Graph) -> List<String>`

  In this function, you will need to get all the keys of the vertices given a list of edges, i.e. a graph.

- `bellman-ford(graph :: Graph, src :: String) -> List<Weight>`

  In this function, you will need to implement the Bellman-Ford algorithm by repeatedly iterating over the edges in the graph and updating a list of weights appropriately.

⋆ **Once a lab TA signs off on your work, you've finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you've just written.**