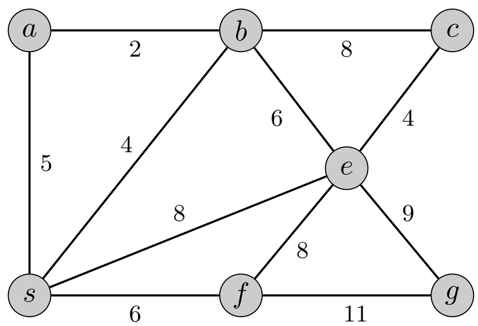
Written Problems. due Mar 10 (Wed), 6.30 pm.

1. **Prim's Algorithm**

   Run Prim's algorithm on the following graph, starting from the node s. Write down the intermediate values of the set S in the execution of Prim's algorithm, along with the final MST. In case of a tie, pick the node that comes first in alphabetical order.



Programming Problem: Implementing Dijkstra due Mar 11 (Thu), 11.59 pm.

In this problem, we investigate how Dijkstra's Algorithm may be used to solve the "shortest paths problem" in the real world. You are provided with data from the National Highway Planning Network (NHPN) in `ps5_dijkstra.zip`. You can learn more about the NHPN at `http://www.fhwa.dot.gov/planning/nhpn/`

This data includes node and link text files from the NHPN. Open `nhpn.nod` and `nhpn.lnk` in a text editor to get a sense of how the data is stored (`datadict.txt` has a more precise description of the data fields and their meanings). You are also provided with a Python module `nhpn.py` containing code to load the text files into Node and Link objects. Read `nhpn.py` to understand the format of the Node and Link objects you will be given.

Additionally, you are provided with tools to help you visualize the output from your algorithms. You can use the `Visualizer` class to produce a KML (Google Earth) file. To view such a file on Google Maps, place it in a web-accessible location, and then search for its URL on Google Maps.

For this problem, you will modify the file `dijkstra.py`. As you solve each part of the problem, check your work by running `test_dijkstra.py`. Submit `dijkstra.py` by copying the file to your `submit` directory.

1. Write a short function `node_by_name(nodes, city, state)` to return a node from the given city/state. Note that some nodes have a description which isn't solely the city name, e.g. `CAMBRIDGE NW` or `NORTH CAMBRIDGE`, either of which we would like to match a query where `city=='CAMBRIDGE'`. Given a choice of more than one node, choose the first node that appears in the data.

2. The links you are given do not include weights, so instead we will use the geographical positions of the edge's nodes.

Write a function `distance(node1, node2)` to return the distance between two NHPN nodes. Nodes come with latitude and longitude (in millionths of degrees). For simplicity, treat these instead as (x, y) coordinates on a flat surface, where the distance between two points can be easily calculated using the Pythagorean Theorem.

Hint: You may find the `math.hypot` function useful.

3. Implement Dijkstra's algorithm to find the shortest path between two vertices in a graph with non-negative edge weights.

   Your function `shortest_path(nodes, edges, weight, s, t)` will be given a list of Node objects, a list of Edge objects (undirected), a function `weight(node1, node2)` which returns the weight of any edge between `node1` and `node2`, a source Node s and a destination Node t. Your function should return a list of `Node` objects representing a path from s to t.

   Dijkstra's algorithm uses a priority queue, but this priority queue has one subtle requirement not met by the `heap.py` implementation seen in Homework 2. Dijkstra's algorithm calls `decrease_key`, but `decrease_key` requires the index of an item in the heap, and Dijkstra's algorithm would have no way of knowing the current index corresponding to a particular Node. To solve this problem, you are provided with an augmented heap object, `heap_id`, with the following extra features:

   - `insert(key)` returns a unique ID.
   - A new method, `decrease_key_using_id(ID, key)` takes an `ID` instead of an index.
   - A new method, `extract_min_with_id()` extracts the minimum element and returns a pair `(key, ID)`

   You may `import heap_id`, without submitting the separate file.

   Hint: The format in which you are given the data (a list of nodes, and a list of edges), is not what you want to use for Dijkstra's algorithm. Start by preprocessing the data into a more useful graph representation. Don't forget that the edges you are given are undirected.

4. **(Optional)** Included in `nhpn.py` is a method to convert a list of nodes to a `.kml` file. `.kml` files can be viewed using Google Maps, by putting the file in a web-accessible location, going to `http://maps.google.com` and putting the URL in the search box.

   Run `visualize_path.py`. This will create two files, `path_flat.kml` and `path_curved.kml`. Both should be paths from Pasadena CA to Cambridge MA. `path_flat.kml` was created using the distance function you wrote in part (b), and `path_curved.kml` was created using a distance function that does not assume the Earth is flat. Can you explain the differences? Also, try asking Google Maps for driving directions from Pasadena to Cambridge.