CMPUT 275 — Tangible Computing **Winter 2016**
Assignment 1 — Directions **Version 1.4 (Jan 23, 2016)**

Driving Route Finder

You will be implementing a driving route finder (like Google Maps) for the Edmonton area. The application will involve a combination of the Arduino (as client) and a python app (as server) on a desktop computer. The user will be able to scroll around with a joystick on a zoomable map of Edmonton and select a start and end point for their route. The Arduino client will communicate these points to the python server, which has all of the street information for Edmonton. The server application will find the shortest path (by distance along the path) and return the waypoints of this path to the Arduino client. The Arduino will then display the path as lines overlaid on the original map. The user can then repeatedly query new points via the Arduino to receive new routes.

We will provide the Arduino code for scrollable/zoomable maps, which will just display the latitude and longitude of a selected point. In class we will develop python code for directed graphs, and discuss algorithms for route finding efficiently. We will also provide you with a text file containing information on the Edmonton road graph. You are responsible for implementing the following (more details on each task will follow):

1. Routines for building the graph by reading the provided text file.
2. A cost function for computing edge weights.
3. A routine for computing least cost paths efficiently.
4. A python route finding server that accepts route finding requests and provides paths through the specified protocol. For part 1, your app will communicate via stdin and stdout. For part 2, it will communicate with your Arduino.
5. Augment the provided Arduino client program so that it sends a request to the route finding server and then receives the route data from the server.
6. Displaying the route on the Arduino overlaid on the map.

The assignment must be submitted in two parts. **The first part is due Monday, February 8. The second part is due Monday, February 22**. Both must be submitted by 11:55pm. You can submit multiple times; only the last submission will be graded. You should submit early to avoid being cutoff because the server is overloaded.

If you're looking for some optional extra functionality for your program after this basic functionality is completed, consider connecting the route finder with our restaurant finder to find paths to nearby restaurants.

Part I: Server

For part I, you will implement the server side of the required functionality (items 1-4 in the list above).

Dijkstra's Algorithm

You will need to implement Dijkstra's algorithm for finding least cost paths matching the following interface.

```
def least_cost_path(graph, start, dest, cost):
    """Find and return the least cost path in graph from start
    vertex to dest vertex.

    Efficiency: If E is the number of edges, the run-time is
      O( E log(E) ).

    Args:
      graph (Graph): The digraph defining the edges between the
        vertices.
      start: The vertex where the path starts. It is assumed
        that start is a vertex of graph.
      dest:  The vertex where the path ends. It is assumed
        that start is a vertex of graph.
      cost:  A function, taking the two vertices of an edge as
        parameters and returning the cost of the edge. For its
        interface, see the definition of cost_distance.

    Returns:
      list: A potentially empty list (if no path can be found) of
        the vertices in the graph. If there was a path, the first
        vertex is always start, the last is always dest in the list.
        Any two consecutive vertices correspond to some
        edge in graph.

    >>> graph = Graph({1,2,3,4,5,6}, [(1,2), (1,3), (1,6), (2,1),
            (2,3), (2,4), (3,1), (3,2), (3,4), (3,6), (4,2), (4,3),
            (4,5), (5,4), (5,6), (6,1), (6,3), (6,5)])
    >>> weights = {(1,2): 7, (1,3):9, (1,6):14, (2,1):7, (2,3):10,
            (2,4):15, (3,1):9, (3,2):10, (3,4):11, (3,6):2,
            (4,2):15, (4,3):11, (4,5):6, (5,4):6, (5,6):9, (6,1):14,
            (6,3):2, (6,5):9}
    >>> cost = lambda u, v: weights.get((u, v), float("inf"))
    >>> least_cost_path(graph, 1,5, cost)
    [1, 3, 6, 5]
    """
```

We will present pseudocode for the algorithm in class.

Graph Building

Your server on start-up will need to load the Edmonton map data into a digraph object, and store the ancillary information about street names and vertex locations. The locations will be used in this assignment. While in this assignment we will not use the street names, a later exercise will use them, hence the need to store them.

The data is stored in a CSV format in a file called `edmonton-roads-2.0.1.txt`, available from the course webpage. An excerpt of the file looks as follows:

```
V,29577354,53.430996,-113.491331
V,1503281720,53.434340,-113.490152
V,36396914,53.429491,-113.491863
E,36396914,29577354,Queen Elizabeth II Highway
E,29577354,1503281720,Queen Elizabeth II Highway
```

In the file there are two types of lines: Those starting with `V` (standing for 'vertex') and those starting with `E` (standing for 'edge'). In a line that starts with `V`, you will find a unique vertex identifier followed by two coordinates, giving the latitude and longitude of the vertex (in degrees). In a line that starts with `E`, you will find the identifiers of the two vertices that are connected by the edge, followed by the street name along the edge.

Two extra requirements:

1. You must store the coordinates in $100,000$ths of a degree as integers (the client will use this convention, too). If the number red in is in variable $coord$, convert it to the integer using $int(float(coord) * 100000)$.
2. You must use the identifiers read from the file and converted to integers as the vertex identifiers in the graph. This is necessary so that we can test your code.

Cost Function

You will also need to write a cost function for use with route finding. This cost function should match this interface.

```python
def cost_distance(u, v):
    '''Computes and returns the straight-line distance between the two
    vertices u and v.

    Args:
        u, v: The ids for two vertices that are the start and
          end of a valid edge in the graph.

        Returns:
          numeric value: the distance between the two vertices.
    '''
```

If you compute the straight-line distance (that is, the ℓ^2 distance, or Euclidean distance) directly using the stored lat and long, then the distance will be in units of $100,000$ths of a degree.

Consider that by using a cost function like this, we could later easily extend this application to allow the user to select between a variety of distance metrics, like Manhattan Distance.

Server

Your server needs to provide routes based on requests from clients. For Part I, your server will be receiving and processing requests from the keyboard, by reading and writing to `stdin` and `stdout`. For Part 2, your server will be communicating with your Arduino. In both cases we will use the same protocol described below, the only change will be that the server will be reading to and from the serial port connected to the Arduino (how to do this will be described later).

All requests will be made by simply providing a latitude and longitude (in $100,000$ths of degrees) of the start and end points in ASCII, separated by single spaces and terminated by a newline. The line should start with the character `R`. For example,

```
R 5365486 -11333915 5364728 -11335891<\n>
```

is a valid request sent to the server (the newline character is shown with `<\n>`). The server will then process the request by first finding the closest vertices in the roadmap of Edmonton to the start and end points (calculate the Euclidean Distance as you did in `cost_distance`), next computing a shortest path along Edmonton streets between the two vertices found, and finally communicating the found waypoints from the first vertex to the last back to the Arduino. The communication of the waypoints to the Arduino is done by a series of prints, each of which consists of the latitude and longitude of a waypoint along the path. However, before telling the first waypoint, the server tells the client the number of waypoints. After each print, the Arduino must acknowledge the receipt of data (preventing unwanted buffer overflows). As an example, assume that the number of waypoints is 8. Then, the server starts with

```
N 8<\n>
```

and the Arduino responds with

```
A<\n>
```

Next, the server sends the coordinates of the first waypoint (corresponding to the location of the first vertex):

```
W 5365488 -11333914<\n>
```

The client responds again with

```
A<\n>
```

Upon receiving this acknowledgement, the server sends the next waypoint, which the client acknowledges again. This continues until there are no more waypoints, when the server sends the character 'E' followed by a newline:

```
E<\n>
```

This ends the session between the client and the server: The server's next state is to wait for the next request from the client, and the client (to be implemented in Part II) will show the route and allow the user to select new start and end points.

By showing the data sent by the server in black and the data sent by the client in blue, the above exchange of messages looks as follows:

```
N 8<\n>
A<\n>
W 5365488 -11333914<\n>
A<\n>
W 5365238 -11334423<\n>
A<\n>
W 5365157 -11334634<\n>
A<\n>
W 5365035 -11335026<\n>
A<\n>
W 5364789 -11335776<\n>
A<\n>
W 5364774 -11335815<\n>
A<\n>
W 5364756 -11335849<\n>
A<\n>
W 5364727 -11335890<\n>
A<\n>
E<\n>
```

The number of spaces between the letters and numbers in all cases is one.

When there is no path from the start to the end vertex nearest to the start and end points sent to the server, the server should return an empty path to the Arduino by sending the "no path" message

```
N 0<\n>
```

The Arduino upon receiving this message should notify the user that there is no path on the road network from the start to the end and should allow the user the select a new pair of start and end points. In particular, the Arduino does not need to acknowledge the receipt of the "no path" message. Accordingly, upon sending the "no path" message, the server should consider the answer to the Arduino's request complete, and move to the state where it waits for the next request.

After loading the Edmonton map data the server should only begin processing requests if it is running as the main program. For example,

```
if __name__ == "__main__":
    # Code for processing route finding requests here
```

This will allow us to test your code for `least_cost_path` and `cost_distance` separately from the server protocol.

Part II: Client

For part II, you will implement the Arduino side of the assignment (items 4-5 above) and complete the communication protocol.

We will provide you with code for moving around with a joystick on a scrollable/zoomable map of Edmonton. You will have to implement the interface for selecting two points with the joystick, communicating these points to the server over the serial port, receiving the resulting path and displaying the path overlaid on the map. While the route is displayed, the user should be able to continue to scroll/zoom on the map. If they select new start and destination points, a new path should be retrieved from the server. You may decide whether you want to still display the route when the user has already selected a start point but has not yet selected a destination point.

For the communication protocol, the server must switch to communicating with the Arduino through a serial port via the help of the `pyserial` package. Files will be provided to show how to use this module and also to test the communication with the Arduino.

In addition, both the client and the server must implement timeouts when waiting for a reply from the other party. In particular, the timeout should be effective when the server is waiting for acknowledgement of data receipt from the Arduino, and also when the Arduino is waiting for either the number of waypoints, the next waypoint, or the final 'E' character. The length of the timeout is by default one second, except when the Arduino waits for the number of waypoints to be received from the server, in which case the timeout should be 10 seconds. When a timeout expires, the server should reset its state to waiting for a client to start communicating with it. Similarly, when a timeout expires, the Arduino should restart the communication attempt. Both the client and the server should similarly reset their states upon receiving a message which does not make sense in their current state.