CMPUT 275 — Tangible Computing **Winter 2018**
Assignment 1 — Directions

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 class 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 12. The second part is due Monday, February 26**. 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.

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 a 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 dest is a vertex of graph.
      cost:  A class with a method called "distance" that takes
        as input an edge (a pair of vertices) and returns the cost
        of the edge. For more details, see the CostDistance class
        description below.

    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.
    """
```

We will present pseudocode and step through an example for the algorithm in class.

Graph Building

Upon starting up, your server will need to load the Edmonton map data into a *digraph* object, and store the ancillary information about vertex locations.

The data is stored in 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.

Promises: No two vertex lines have the same identifier. The endpoints of an edge will be specified on previous lines before the line describing the edge. Noe street name has a comma.

Two extra requirements:

1. You must store the coordinates in 100,000-ths of a degree as integers (the reason being that the client will use this convention, too). If you read a coordinate, such as 53.430996, into variable $coord$, convert it to the integer to be stored in the graph by using $int(float(coord) * 100000)$. In this example you would store 5343099.
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.

In particular, you must implement the following function:

```
def load_edmonton_graph(filename):
    """
    Loads the graph of Edmonton from the given file.

    Returns two items
      graph: the instance of the class Graph() corresponding to the
        directed graph from edmonton-roads-2.0.1.txt
      location: a dictionary mapping the identifier of a vertex to
        the pair (lat, lon) of geographic coordinates for that vertex.
        These should be integers measuring the lat/lon in 100000-ths
        of a degree.

    In particular, the return statement in your code should be
      return graph, location
    (or whatever name you use for the variables).

    Note: the vertex identifiers should be converted to integers
      before being added to the graph and the dictionary.
    """
```

When the server calls this function, you should call it with filename being `"edmonton-roads-2.0.1.txt"`.

Note we are not using the street names in this assignment, they can be ignored.

Cost Function Class

You will also need to write a cost function for use with route finding. The function will take the identifiers of two vertices and return the standard Euclidean distance between the two. However, this requires the cost function to have access to the latitude and longitude of the two endpoints.

So you will create a class with a method called distance. The reason we use a class is that we can pass it the dictionary mapping vertices to their geographic coordinates and then the distance method can use this dictionary.

The class should match this description precisely. You are in charge of implementing the methods.

```
class CostDistance:
    """
    A class with a method called distance that will return the Euclidean
    between two given vertices.
    """

    def __init__(self, location):
        """
        Creates an instance of the CostDistance class and stores the
```

```
        dictionary "location" as a member of this class.
        """

    def distance(self, e):
        """
        Here e is a pair (u,v) of vertices.
        Returns the Euclidean distance between the two vertices u and v.
        """
```

Recall that the Euclidean distance between two points (x_1, y_1) and (x_2, y_2) is

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Consider that by using an approach 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). Thus, while the description below talks about the server communicating with the Arduino, for Part 1 the Arduino will not be there and the server will be reading and printing to to `stdin` and `stdout` as noted above.

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 (break ties arbitrarily), 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.

- Part I: At this point your program should end. It will process only one request.
- Part II: 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:

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

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 `CostDistance` separately from the server protocol.

For Part I, it is up to you how you implement error handling.

Submission

The server should be implemented in file `server.py` and by importing this file, it should be possible use the functions that you had to write. That is, the following code should succeed when run from a script put next to your file `server.py`:

```
import server
from graph import Graph
# first load the edmonton graph
edmonton_graph, location = server.load_edmonton_graph("edmonton-roads-2.0.1.txt")
# create the CostDistance instance
cost = server.CostDistance(location)
# now test the distance method from CostDistance
print(cost.distance((1503281720, 29577354)))
# build a test 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)])
# lengths of the edges described explicitly
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}
# a simple "dummy" class that just uses the weights in the above dictionary
class SimpleDist:
    def distance(self, e): return weights[e]
cost2 = SimpleDist()
# test the least_cost_path function
print(server.least_cost_path(graph, 1, 5, cost2))
```

In particular, this code should print

```
355.1746049480452
```

```
[1, 3, 6, 5]
```

It is ok if the actual distance is just very close (say, within 10^{-8}) to this one. This may be the case due to rounding issues.

Finally: We should be able to run the entire server using

```
python3 server.py
```

This will load the graph of edmonton from `"edmonton-roads-2.0.1.txt"`, read a request from `sys.stdin` and print the output to `sys.stdout`.

You can test your entire server using some of the files on eClass. For example:

```
python3 server.py < test00-input.txt > mysol.txt
```

This will cause `sys.stdin` to read from `test00-input.txt` instead of the keyboard and have `sys.stdout` print to the file `mysol.txt` instead of the terminal. You can examine the output by looking at `mysol.txt`. You can quickly determine if the output agrees with with the provided expected output `test00-output.txt` by running

```
diff mysol.txt test00-output.txt
```

These test files are in the `tests.tar.gz` file on eClass. There is also a script `test_server.py` that includes the Python code for testing individual functions in the example above. The expected output for this is in `test_server_output.py`.

Part II: Client

For part II, you will implement the Arduino side of the assignment (items 4-5 above) and complete the communication protocol. A video demonstrating this part will be uploaded sometime before the deadline for part 1.

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 `serial` 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.

Additional Details

You can view your three main tasks as follows:

- Modify the Python server so it communicates over the serial port. An example of how to do this is in the Google Drive directory for the course, in the `lect09-feb06` directory. Recall that the `Serial` object reads and writes byte arrays to the serial port. See the example in Google Drive to see how to convert between byte arrays and normal strings.

  Note, in particular, that `Serial.readline()` will read up to and including the `\n` character and include this character in the byte array. But `Serial.write()` does not write a `\n` character itself, you will have to explicitly send this character when you need to.

  You must use a timeout of 1 second anytime you want to read from the serial port. After a timeout or anytime the server reads an unexpected message, it should return to the state where it is waiting for the initial request.

  As stated above, if there is no path in the Edmonton road graph, simply communicate a message of `N 0\n` and return to the state where it is waiting for a request. Do not wait for an acknowledgement in this case.

- Have the Arduino communicate with the server to get the waypoints. The Arduino should use exactly the protocol described above.

  You should use a timeout of 10 seconds when waiting for the first reply of the form
  `N <num_waypoints>\n`
  as the server may take a few seconds to compute this path using Dijkstra's algorithm. You should use a timeout of 1 second for every part of the communication (i.e. when waiting for lines starting with `W` or `E`).

  After a timeout or anytime the client reads an unexpected message, it should restart the request from the beginning (i.e. starting with the `R` line).

  The code provided allows up to 500 waypoints to be stored. This is large enough to store almost all possible routes but small enough to give you a few hundered extra bytes of memory to work with. If the path being communicated has over 500 waypoints, your client will treat this as no path. You can handle this however you want: read all waypoints but don't store them, or just quit and let the server timeout waiting for the acknowledgement, or anything else that works.

- Drawing the waypoints. If there is a path, you should draw it anytime you update the map (and after the path is first received). Use the drawLine method of the tft object. See the video for some small details you may wonder about.

Misc. Notes

- You may notice the scrolling slows down a lot, especially in the highest zoom level near the lower-right corner of the map. This is ok and it is not your job to fix. This is because redrawing the patch of the map after moving the cursor has to seek to near the end of a really big .lcd file.

  A way around this could be to write the images to known blocks of memory (like with the restaurant data) so we can index into the correct position directly, but this is much more complicated so this simpler solution suffices.

- It is ok if your cursor erases part of the route you drew. Of course, there are ways to avoid this but it is not a required part of the assignment.

- To make the search run faster for queries near each other, you may terminate it as soon as the endpoint is added to the reached dictionary. It won't be a dictionary of all reached vertices, but it has the endpoint vertex of interest. **This is optional**.

  Note, it is not sufficient to terminate the search the first time an edge is added to the heap where the endpoint is the destination. This may not be the edge that reaches the destination first. So only quit after the destination is added to the reached dictionary!

- Read and understand the provided client code! It is possible to finish this assignment by creating some helpful functions and calling them from the `TODO` comments in `client.cpp`, but it certainly helps to understand the code base in its entirety. You may even add additional files if it helps.

- Indicate in the `README` every place where the code was modified! This may be slightly tedious, but it is required. We don't need an exhaustive description of what was changed (it should be clear from your comments), we just need to know where to look.

- If you spot a bug in the provided code, you are not required to fix it but the instructors would still like to know about it!