Tim Leggas and Jason Qiu

1: User Manual

If there errors, then please try quitting the kernel and running it again.

To run the project, go to Evaluation > Evaluate Notebook. A dialogue window will pop up with three buttons -- “Add Locations”, “Navigate”, and “Quit”.
Click “Quit” to exit out of the project.

In order to add locations to route, click “Add Locations” first.
The first window will close, and another window will pop up with three buttons -- “Use Address”, “Use Map”, and “Return”.
Click “Return” to go back to the first window.

To add locations directly using addresses, click “Use Address”.
The “Add Locations” window will close, and another window will pop up with an input field and two buttons -- “Add Address” and “Return”.
Input an address into the input field (make sure to include the city; e.g. “1234 Main Street, Bowling Green”) and click “Add Address” to add it to the route.
Click “Return” to go back to the “Add Locations” window.

To add locations by clicking on a map, click “Use Map”.
The “Add Locations” window will close, and another window will pop up with a map of Bowling Green, instructions, and three buttons -- “Add Location”, “Name Select”, and “Return”.
First, click “Add Location”. Another window will pop up with a draggable, zoomable, clickable map of Bowling Green and “Return” button.
After clicking on a location to add, click “Return” to go back to the previous window.
Finally, click “Name Select” to add a name to the location that was clicked as well as the time you have to get to the location from when you start driving from the first location of the route, and it will be added to the route.
Click “Return” to go back to the “Add Locations” window.

In order to preview, delete from, and navigate the route, click “Navigate” first.
The first window will close, and another window will pop up with a map and 5 buttons -- “Previous”, “Next”, “Delete Location”, “Navigate”, and “Return”.

To view the locations in the route, click the respective “Previous” or “Next” buttons to go forward or backward through the route.

To delete locations from the route, find the location to be deleted using the “Previous” and “Next” buttons and click “Delete Location” to remove it from the route.

To calculate the shortest path to get to each location of the route in order, click “Navigate”.
Another window will pop up, with a “Start” button. Click “Start”.
The window will close, and another window will pop up, displaying “Please Wait” while the path is being calculated (this may take some time).
After the calculations are finished, the “Please Wait” window will close and another window will pop up, showing a map with some points and lines of various colors, and a legend.
Each distinctly-colored line represents the shortest path from one of the given locations to the next.

3: Algorithms

Dijkstra’s Algorithm

The most important algorithm used in the project is Dijkstra’s Algorithm.

Dijkstra’s Algorithm takes a weights graph (in our case, the graph represents a road network, the vertexes represent intersections, and the edge weights represent distance), start vertex id, and end vertex id and returns a list of vertexes that is the shortest path from the start vertex to the end vertex.

```
(* Pseudocode *)
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
10     dist[source] ← 0
```

First, it creates a list of unvisited vertexes called *Q*, initializes all values in *dist*, an association map used to store the shortest distance from the start vertex to each vertex, to infinity, except the start vertex, which is set to 0 (because the distance from the start vertex to itself is 0), and sets the value of the start vertex in *prev*, an association map used to store the neighboring vertex of a vertex *u* that vertex *u* should follow in order to take the shortest path to the start vertex, to itself (because the vertex that is closest to the start vertex from the start vertex is itself).

```
(* Psuedocode *)
    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
```

Next, while *Q* still has unvisited vertexes, choose *v*, the one with the smallest value from dist (the current shortest path), remove it from *Q* (since we are now visiting it), and then find its neighbors (the vertexes that it is connected to). For each neighbor *v* of that *u*, find *alt*, the distance from it to start using the current shortest path. If *alt* is less than the value of *v* in *dist*, then assign *alt* to *v* in *dist* (because the *alt* path is quicker than the previous path that the value of *v* in *dist* was calculated from) and assign *u* to *v* in *prev* (because at the moment, *u* is the closest neighbor of *v* to the start vertex).

```
(* Pseudocode *)
 S ← empty sequence
 u ← target
   while u is defined:                      // Construct the shortest path with a stack S
       insert u at the beginning of S       // Push the vertex onto the stack
       u ← prev[u]                           // Traverse from target to source

 return S,dist[],prev[]
```

Finally, after all nodes' shortest distances from the start vertex are calculated, getting the actual shortest path from any vertex to the start vertex is extremely simple: Push the end vertex onto a stack *S*, find the value of the desired end vertex in *prev* and push it onto *S*, find the value of that value in *prev* and push it onto *S*, and so on until that value equals the start vertex and is pushed onto *S*.

S contains the completed shortest path; if it is continuously popped until empty, then it will return each vertex that must be travelled in order to take the shortest path to whatever end vertex was chosen.

dist contains each vertex's shortest distance from that vertex to the start vertex.

prev contains each vertex's neighbor that is closest to the start vertex.

4: External Files

The project imports a .graphml files, which create graphs. GraphML files are like XML files, except they are sort of object-oriented. For example, each node (vertex) and edge is declared with various

attributes:

```
...
<node id="168710743">
    <data key="d3">nan</data>
    <data key="d4">168710743</data>
    <data key="d5">547578.5293574127</data>
    <data key="d6">4093756.4937942335</data>
    <data key="d7">-86.465347</data>
    <data key="d8">36.988737</data>
 </node>
...
```

```
...
<edge id="0" source="168630520" target="168813025">
    <data key="d9">16384175</data>
    <data key="d13">Normalview Drive</data>
    <data key="d10">residential</data>
    <data key="d11">False</data>
    <data key="d12">52.899</data>
 </edge>
...
```

The keys are aliases for various attributes declared at the top of the file:

```
...
<key attr.name="name" attr.type="string" for="edge" id="d13"/>
<key attr.name="length" attr.type="string" for="edge" id="d12"/>
<key attr.name="oneway" attr.type="string" for="edge" id="d11"/>
...
```

7: External Code

We copied code from *https://mathematica.stackexchange.com/questions/59543/convert-locator-x-y-values-to-lat-lon-as-per-getcoordinates-on-map* in order to convert locator coordinates into latitude and longitude.

We used a Python library from *https://geoffboeing.com/publications/osmnx-complex-street-networks* in order to convert the .osm files containing the road data that we downloaded into .graphml files that were (1) more easy to extract information (vertex coordinates and edge weights) from and (2) able to be imported into Mathematica (although only vertex ids are recognized).

8: Limitations and Issues During Development

Limitations

The project cannot process large areas very quickly, so the demo version contains a graph only a fourth the size of Bowling Green. The boundaries of where paths can be calculated are about 36.9908N,

36.9586S, -86.4478E, -86.4886W.

The open-source data the project uses can be a bit spotty for less popular roads, but it still contains the majority of all roads and definitely contains main roads.

The built-in DynamicGeoGraphics function in Mathematica is experimental, so it can be somewhat buggy in that it sometimes takes an extremely long time to display map background tiles; if you zoom in far enough, it will likely eventually load.

Addresses when inputted should be fairly specific (should include address and city name at the very least) but can also automatically interpret some places in the place of addresses, such as Walmart and Kroger.

Issues During Development ***(add QGIS and file extension confusion)***

In the beginning, we had a very difficult time coming up with a solution. The first one we tried was image processing: taking screenshots of the built-in GeoGraphics maps in Mathematica, recognizing/isolating the roads in the image from everything else, and then converting it into a graph somehow. We were still in the process of figuring out how to process the images in order to isolate the roads when the file we were working on became corrupted and our work was essentially lost, since it consisted largely of almost-randomly cobbling image processing functions together using, again, almost-random magic numbers until certain roads were isolated.

After that, we searched around on the Internet and decided to take a more data-focused approach as opposed to images. We found open-source street data and used Java to extract the data that we needed to create a graph from it. The reason we used Java was because it was instant, speed-wise, as opposed to Mathematica, which we had little confidence in (1) being able to quickly create code for that could extract specific data and (2) being quick enough to try many different approaches in a small timeframe. Furthermore, we do not believe that it is the core of what we are trying to accomplish with the project -- finding the shortest distance between two places while following roads -- so we are fine with not using Mathematica to do it.

However, we soon found that Mathematica 11.3 did not take a liking to creating large graphs with very precise vertex coordinates, in that it would give a very strange error. Although this was resolved after updating to Mathematica 12, the update also broke many components of the already nearly-completed user interface, and the graph it created was undirected, meaning that one-way streets would not function properly.

At around the same time, we found a Python library that would very conveniently convert the street data that we downloaded into a format that Mathematica could interpret (with directed edges as well);

however, it would only recognize the edges and vertex ids, so the other elements that we required, vertex coordinates and edge weights, were not present. We examined the file that the Python library outputted and found that the information we needed was in it, only it was formatted in a way that Mathematica could not recognize. After some small tweaks to the file using Java and manual editing, Mathematica seemed like it could recognize the file, but it also seemed like it would take an eternity to find out (the import never finished running).

Since that didn't work, we decided to simply extract the information we needed from the file using Java and add it back to the information-less graph once it was imported into Mathematica, which, thankfully, ended up working.

Next, we needed to implement Dijkstra's algorithm, which we ran into some trouble with but eventually figured out. In addition, we managed to get it to work with our directed graphs.

9: References

We downloaded the street data from *https://www.openstreetmap.org.*

We used the pseudocode at *https://en.wikipedia.org/wiki/Dijkstra's_algorithm* to implement Dijkstra's Algorithm into Mathematica.