cuGraph Single Source Shortest Path

In this notebook, you will use GPU-accelerated graph analytics with cuGraph to identify the shortest path from node on the road network to every other node, both by distance, which we will demo, and by time, which you will implement. You will also visualize the results of your findings.

Presentation

Please execute the cell below to watch the instructor presentation before proceeding with the rest of the notebook.

In [ ]:
```
%%html
<video width="800" controls>
  <source src="https://dli-lms.s3.us-east-1.amazonaws.com/assets/s-ds-01-
</video>
```

Objectives

By the time you complete this notebook you will be able to:

- Use GPU-accelerated SSSP algorithm
- Use cuXfilter to create a heatmap of average travel times

Imports

In [ ]:
```
import cudf
import cugraph as cg

import cuxfilter as cxf
from bokeh.palettes import Magma, Turbo256, Plasma256, Viridis256
```

Loading Data

We start by loading the road graph data you prepared for constructing a graph with cuGraph, with the long unique `nodeid` replaced with simple (and memory-efficient) integers we call the `graph_id`.

In [ ]:
```
road_graph = cudf.read_csv('./data/road_graph_2-09.csv', dtype=['int32',
road_graph.head()
```

Next we load the graph-ready data you prepared that uses amount of time traveled as edge weight.

In [ ]:
```
speed_graph = cudf.read_csv('./data/road_graph_speed_2-09.csv', dtype=['i
speed_graph.head()
```

Finally we import the full `road_nodes` dataset, which we will use below for visualizations.

In [ ]:
```
road_nodes = cudf.read_csv('./data/road_nodes_2-09.csv', dtype=['str', 'f
road_nodes = road_nodes.drop_duplicates() # again, some road nodes appear
road_nodes.head()
```

In [ ]:
```
road_nodes.shape
```

In [ ]:
```
speed_graph.src.max()
```

Construct Graph with cuGraph

Now that we have the well-prepped `road_graph` data, we pass it to cuGraph to create our graph datastructure, which we can then use for accelerated analysis. In order to do so, we first use cuGraph to instantiate a `Graph` instance, and then pass the instance edge sources, edge destinations, and edge weights, currently the length of the roads.

In [ ]:
```
G = cg.Graph()
%time G.from_cudf_edgelist(road_graph, source='src', destination='dst', e
```

Analyzing the Graph

First, we check the number of nodes and edges in our graph:

In [ ]:
```
G.number_of_nodes()
```

In [ ]:
```
G.number_of_edges()
```

We can also analyze the degrees of our graph nodes. We would expect, as before, that every node would have a degree of 2 or higher, since undirected edges count as two edges (one in, one out) for each of their nodes.

In [ ]:
```
deg_df = G.degree()
deg_df['degree'].describe()[1:]
```

We would also expect that every degree would be a multiple of 2, for the same reason. We check that there are no nodes with odd degrees (that is, degrees with a value of 1 modulo 2):

In [ ]:
```
deg_df[deg_df['degree'].mod(2) == 1]
```

Observe for reference that some roads loop from a node back to itself:

In [ ]:
```
road_graph.loc[road_graph.src == road_graph.dst]
```

Single Source Shortest Path

To demo the Single Source Shortest Path (SSSP) algorithm, we will start with the node with the highest degree. First we obtain its `graph_id`, reported by the `degree` method as `vertex`:

In [ ]:
```
demo_node = deg_df.nlargest(1, 'degree')
demo_node_graph_id = demo_node['vertex'].iloc[0]
demo_node_graph_id
```

We can now call `cg.sssp`, passing it the entire graph `G`, and the `graph_id` for our selected vertex. Doing so will calculate the shortest path, using the road length weights we have provided, to *every* other node in the graph - millions of paths, in seconds:

In [ ]:
```
%time shortest_distances_from_demo_node = cg.sssp(G, demo_node_graph_id)
shortest_distances_from_demo_node.head()
```

In [ ]:
```
# Limiting to those nodes that were connected (within ~4.3 billion meters
# cg.sssp uses the max int value for unreachable nodes, such as those on
shortest_distances_from_demo_node['distance'].loc[shortest_distances_from
```

Exercise: Analyze a Graph with Time Weights

For this exercise, you are going to analyze the graph of GB's roads, but this time, instead of using raw distance for a road's weights, you will be using how long it will take to travel along the road.

Step 1: Construct the Graph

Construct a cuGraph graph called `G_ex` using the sources and destinations found in `speed_graph`, along with length in seconds values for the edges' weights.

In [ ]:
```

```

Solution

In [ ]:
```
%load solutions/construct_graph
```

Step 2: Get Travel Times From a Node to All Others

Choose one of the nodes and calculate the time it would travel to take from it to all other nodes via SSSP, calling the results `ex_dist`.

In [ ]:
```

```

Solution

In [ ]:
```
%load solutions/travel_times
```

Step 3: Visualize the Node Travel Times

In order to create a graphic showing the road network by travel time from the selected node, we first need to align the just-calculated distances with their original nodes. For that, we use the mapping from `node_id` strings to their `graph_id` integers.

In [ ]:
```
mapping = cudf.read_csv('./data/node_graph_map.csv')
mapping.head()
```

We see that the `sssp` algorithm has put the `graph_id`s in the `vertex` column, so we will merge on that.

In [ ]:
```
ex_dist.head()
```

In [ ]:
```
road_nodes = road_nodes.merge(mapping, on='node_id')
road_nodes = road_nodes.merge(ex_dist, left_on='graph_id', right_on='vert
road_nodes.head()
```

Next, we select those columns we are going to use for the visualization.

For color-scaling purposes, we get rid of the unreachable nodes with their extreme distances, and we invert the distance numbers so that brighter pixels indicate closer locations.

In [ ]:
```
gdf = road_nodes[['east', 'north', 'distance']]
gdf = gdf[gdf['distance'] < 2**32]
gdf['distance'] = gdf['distance'].pow(1/2).mul(-1)
```

Otherwise, this visualization will be largely similar to the scatterplots we made to visualize the population, but instead of coloring by point density as in those cases, we will color by mean travel time to the nodes within a pixel.

In [ ]:
```
cxf_data = cxf.DataFrame.from_dataframe(gdf)
```

In [ ]:
```
chart_width = 600
heatmap_chart = cxf.charts.datashader.scatter(x='east', y='north',
                                              width=chart_width,
                                              height=int((gdf['north'].ma
                                                          (gdf['east'].max
                                                           chart_width),
                                              #palettes=Plasma256, # try
                                              #pixel_shade_type='linear',
                                              aggregate_col='distance',
                                              aggregate_fn='mean',
                                              #point_shape='square',
                                              point_size=1)
```

In [ ]:
```
dash = cxf_data.dashboard([heatmap_chart], theme=cxf.themes.dark, data_si

heatmap_chart.view()
```

In [ ]:
```
%%js
var host = window.location.host;
element.innerText = "'"+host+"'";
```

Set `my_url` in the next cell to the value just printed, making sure to include the quotes:

In [ ]:
```
my_url = # TODO: Set this value to the print out of the cell above, inclu
dash.show(my_url, port=8789)
```

... and you can run the next cell to generate a link to the dashboard:

In [ ]:
```
%%js
var host = window.location.host;
var url = 'http://'+host+'/lab/proxy/8789/';
element.innerHTML = '<a style="color:blue;" target="_blank" href='+url+'>
```

In [ ]:
```
dash.stop()
```

Next

This concludes the second section of the workshop. In the third section, *Project: Biodefense*, you'll put the skills you've learned in this workshop to the test by helping over several simulated days to address a national epidemic.

Optional: Restart the Kernel

If you plan to do additional work in other notebookes, please restart the kernel:

In [ ]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```