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.

Objectives

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

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

Imports

In [1]:
```
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 [2]:
```
road_graph = cudf.read_csv('./data/road_graph_2-09.csv', dtype=['int32', 'int32', 'flo
road_graph.head()
```

Out[2]:

	src	dst	length
0	0	129165	44.0
1	1	1678323	70.0
2	1	2372610	18.0
3	1	2483057	40.0
4	2	2	55.0

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

In [3]:
```
speed_graph = cudf.read_csv('./data/road_graph_speed_2-09.csv', dtype=['int32', 'int32
speed_graph.head()
```

Out[3]:

	src	dst	length_s
0	0	129165	3.280848
1	1	1678323	5.219531
2	1	2372610	1.342165
3	1	2483057	2.982589
4	2	2	4.101060

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

In [4]:
```
road_nodes = cudf.read_csv('./data/road_nodes_2-09.csv', dtype=['str', 'float32', 'flo
road_nodes = road_nodes.drop_duplicates() # again, some road nodes appeared on multipl
road_nodes.head()
```

Out[4]:

	node_id	east	north	type
2589119	id000000F5-5180-4C03-B05D-B01352C54F89	432920.250	572547.375000	road end
1954117	id000003F8-9E09-4829-AD87-6DA4438D22D8	526616.375	189678.390625	junction
873541	id000010DA-C89A-4198-847A-6E62815E038A	336879.000	731824.000000	junction
1346912	id000017A0-1843-4BC7-BCF7-C943B6780839	380635.000	390153.000000	junction
1553458	id00001B2A-155F-4CD3-8E06-7677ADC6AF74	337481.000	350509.687500	junction

In [5]:
```
road_nodes.shape
```

Out[5]:
```
(3078117, 4)
```

In [6]:
```
speed_graph.src.max()
```

Out[6]:
```
3078116
```

Construct Graph with cuGraph

Now that we have the well-prepped `road_graph` data, we pass it to cuGraph to create our graph data structure, 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 [7]:
```
G = cg.Graph()
%time G.from_cudf_edgelist(road_graph, source='src', destination='dst', edge_attr='len
```

```
CPU times: user 85 ms, sys: 60.6 ms, total: 146 ms
Wall time: 145 ms
```

Analyzing the Graph

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

In [8]:
```
G.number_of_nodes()
```

Out[8]:
```
3078117
```

In [9]:
```
G.number_of_edges()
```

Out[9]:
```
3620793
```

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 [10]:
```
deg_df = G.degree()
deg_df['degree'].describe()[1:]
```

Out[10]:
```
mean     4.689990
std      1.913452
min      2.000000
25%      2.000000
50%      6.000000
75%      6.000000
max     16.000000
Name: degree, dtype: float64
```

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 [11]:
```
deg_df[deg_df['degree'].mod(2) == 1]
```

Out[11]:

degree	vertex

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

In [12]:
```
road_graph.loc[road_graph.src == road_graph.dst]
```

Out[12]:

	src	dst	length
4	2	2	55.0
145	62	62	108.0
293	124	124	67.0
471	196	196	26.0
571	240	240	44.0
...	...	...	...
7216602	3077469	3077469	78.0
7216735	3077519	3077519	111.0
7216849	3077567	3077567	69.0
7217091	3077670	3077670	30.0
7217294	3077756	3077756	45.0

23417 rows × 3 columns

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 [13]:

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

Out[13]:

```
652907
```

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 [14]:

```
%time shortest_distances_from_demo_node = cg.sssp(G, demo_node_graph_id)
shortest_distances_from_demo_node.head()
```

```
CPU times: user 12.3 s, sys: 78.6 ms, total: 12.4 s
Wall time: 12.4 s
```

Out[14]:

	distance	vertex	predecessor
0	215398.0	452873	200589
1	147424.0	452874	633387
2	167215.0	452876	1641914
3	211350.0	452893	820362
4	151358.0	453033	2635012

In [15]:
```
# Limiting to those nodes that were connected (within ~4.3 billion meters because
# cg.sssp uses the max int value for unreachable nodes, such as those on different isl
shortest_distances_from_demo_node['distance'].loc[shortest_distances_from_demo_node['d
```

Out[15]:
```
mean    209942.046612
std     137073.103410
min          0.000000
25%     124952.000000
50%     181649.000000
75%     252309.000000
max     868580.000000
Name: distance, dtype: float64
```

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 [20]:
```
G_ex = cg.Graph()
G_ex.from_cudf_edgelist(speed_graph, source='src', destination='dst', edge_attr='lengt
```

Solution

In [21]:
```
# %load solutions/construct_graph
G_ex = cg.Graph()
G_ex.from_cudf_edgelist(speed_graph, source='src', destination='dst', edge_attr='lengt
```

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

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

In [22]:
```
ex_deg = G_ex.degree()
ex_node = ex_deg.nlargest(1, 'degree')

%time ex_dist = cg.sssp(G_ex, ex_node['vertex'].iloc[0])

# limiting to those nodes that were connected (within ~4.3 billion seconds; .sssp uses
ex_dist['distance'].loc[ex_dist['distance'] < 2**32].describe()[1:]
```

```
CPU times: user 3.84 s, sys: 39 ms, total: 3.88 s
Wall time: 3.86 s
```

Out[22]:
```
mean      7416.267095
std       4664.674463
min          0.000000
25%       4478.059570
50%       6439.847168
75%       9051.517578
max      31420.681641
Name: distance, dtype: float64
```

Solution

In [23]:
```
# %load solutions/travel_times
# If you have time, see what the SSSP visualization looks like starting from nodes at
# or one of the end nodes of an especially long edge, or even one of the nodes unreach
ex_deg = G_ex.degree()
ex_node = ex_deg.nlargest(1, 'degree')

%time ex_dist = cg.sssp(G_ex, ex_node['vertex'].iloc[0])

# limiting to those nodes that were connected (within ~4.3 billion seconds; .sssp uses
ex_dist['distance'].loc[ex_dist['distance'] < 2**32].describe()[1:]
```

```
CPU times: user 3.83 s, sys: 32 ms, total: 3.86 s
Wall time: 3.85 s
```

Out[23]:
```
mean      7416.267095
std       4664.674463
min          0.000000
25%       4478.059570
50%       6439.847168
75%       9051.517578
max      31420.681641
Name: distance, dtype: float64
```

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 [24]:
```
mapping = cudf.read_csv('./data/node_graph_map.csv')
mapping.head()
```

Out[24]:

	node_id	graph_id
0	id000000F5-5180-4C03-B05D-B01352C54F89	0
1	id000003F8-9E09-4829-AD87-6DA4438D22D8	1
2	id000010DA-C89A-4198-847A-6E62815E038A	2
3	id000017A0-1843-4BC7-BCF7-C943B6780839	3
4	id00001B2A-155F-4CD3-8E06-7677ADC6AF74	4

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

In [25]:
```
ex_dist.head()
```

Out[25]:

	distance	vertex	predecessor
0	9103.391602	245342	1419706
1	6017.406250	245360	1285396
2	5748.593262	245442	1229393
3	23353.482422	245443	2711504
4	3713.541748	245579	43489

In [26]:
```
road_nodes = road_nodes.merge(mapping, on='node_id')
road_nodes = road_nodes.merge(ex_dist, left_on='graph_id', right_on='vertex')
road_nodes.head()
```

Out[26]:

	node_id	east	north	type	graph_id	distance	vertex	predecessor
0	id00F7FA49-BCE9-44AA-9E4C-FC60DEA32522	433720.00000	339688.000000	junction	11744	2139.858643	11744	2098225
1	id00F80B0C-A951-41A9-9452-5FC8CF11F4A8	405220.68750	601848.750000	junction	11745	12234.989258	11745	1268904
2	id00F80DB7-4E0A-450D-8700-97CC39062F12	334536.84375	395009.187500	junction	11746	4633.409668	11746	1945558
3	id00F8133C-D179-41C7-8B2E-2A3150BB2846	271467.87500	654613.562500	road end	11747	13837.215820	11747	696037
4	id00F814B8-E5C7-431F-8AFF-2240F7208D1B	602086.62500	223463.078125	junction	11748	8759.535156	11748	1913983

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 [27]:
```
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 scatter plots 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 [28]:
```
cxf_data = cxf.DataFrame.from_dataframe(gdf)
```

In [29]:
```
chart_width = 600
heatmap_chart = cxf.charts.datashader.scatter(x='east', y='north',
                                              width=chart_width,
                                              height=int((gdf['north'].max() - gdf['nc
                                                          (gdf['east'].max() - gdf['eas
                                                           chart_width),
                                              #palettes=Plasma256, # try also Turbo256
                                              #pixel_shade_type='linear', # can also b
                                              aggregate_col='distance',
                                              aggregate_fn='mean',
                                              #point_shape='square',
                                              point_size=1)
```

In [30]:
```
dash = cxf_data.dashboard([heatmap_chart], theme=cxf.themes.dark, data_size_widget=Tru

heatmap_chart.view()
```

Out[30]:

In [31]:
```
%%js
var host = window.location.host;
element.innerText = "'http://"+host+"'";
```

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

In [33]:
```
my_url = 'http://dli-604a4aa51b37-32d463.aws.labs.courses.nvidia.com'
dash.show(my_url + "/lab", port=8789)
```

```
Dashboard running at port 8789
```

Out[33]:

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

In [34]:
```
%%js
var host = window.location.host;
var url = 'http://'+host+'/lab/proxy/8789/';
element.innerHTML = '<a style="color:blue;" target="_blank" href='+url+'>Open Dashboar
```

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

Next

This concludes the second section of the workshop. In the third section, 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 notebooks, please restart the kernel:

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