Preparing Data for Graph Construction

As part of our larger data science goal for this workshop, we will be working with data reflecting the entire road network of Great Britain. We have as a starting point road data extracted into tabular csv format from official GML files. Ultimately, we would like to use cuGraph to perform GPU-accelerated graph analytics on this data, but in order to do so, we need to do some preprocessing to get it ready for graph creation.

In this notebook you will be learning additional cuDF data transformation techniques in a demonstration of prepping data for ingestion by cuGraph. Next, you will do a series of exercises to perform a similar transformation of the data for the creation of a graph with different edge weights.

Objectives

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

- Create a GPU-accelerated graph
- Perform GPU-accelerated dataframe merge operations with cuDF

Imports

In addition to `cudf` , for this notebook we will also import `cugraph` , which we will use (after data preparation) to construct a GPU-accelerated graph. We also import `networkx` for a brief performance comparison later on.

In [1]:
```
import cudf
import cugraph as cg

import networkx as nx
```

Read Data

In this notebook we will be working with two data sources that will help us create a graph of the UK's road networks.

UK Road Nodes

The first data table describes the nodes in the road network: endpoints, junctions (including roundabouts), and points that break up a long stretch of curving road so that it can be mapped correctly (instead of as a straight line).

The coordinates for each point are in the OSGB36 format we explored earlier in section 1-05.

In [2]:
```
road_nodes = cudf.read_csv('./data/road_nodes_1-06.csv')
road_nodes.head()
```

Out[2]:

	node_id	east	north	type
0	id02FE73D4-E88D-4119-8DC2-6E80DE6F6594	320608.0938	870994.0000	junction
1	id634D65C1-C38B-4868-9080-2E1E47F0935C	320628.5000	871103.8125	road end
2	idDC14D4D1-774E-487D-8EDE-60B129E5482C	320635.4688	870983.9375	junction
3	id51555819-1A39-4B41-B0C9-C6D2086D9921	320648.7188	871083.5625	junction
4	id9E362428-79D7-4EE3-B015-0CE3F6A78A69	320658.1875	871162.3750	junction

In [3]:
```
road_nodes.dtypes
```

Out[3]:
```
node_id     object
east       float64
north      float64
type        object
dtype: object
```

In [4]:
```
road_nodes.shape
```

Out[4]:
```
(3121148, 4)
```

In [5]:
```
road_nodes['type'].unique()
```

Out[5]:
```
0       junction
1    pseudo node
2       road end
3     roundabout
Name: type, dtype: object
```

UK Road Edges

The second data table describes road segments, including their start and end points, how long they are, and what kind of road they are.

In [6]:
```
road_edges = cudf.read_csv('./data/road_edges_1-06.csv')
road_edges.head()
```

Out[6]:

	src_id	dst_id	length	type	form
0	#id138447A5-91D4-4642-BFAC-13F309705429	#id84C9DAD4-9243-4742-B582-E8CBC848E08A	314	Restricted Local Access Road	Single Carriageway
1	#idD615F9C5-5BE9-412D-9FED-F4928BAB4146	#idA1BB20B9-0751-4B42-9925-20607ABF5027	104	Restricted Local Access Road	Single Carriageway
2	#idDC14D4D1-774E-487D-8EDE-60B129E5482C	#id51555819-1A39-4B41-B0C9-C6D2086D9921	100	Restricted Local Access Road	Single Carriageway
3	#id626FC567-199C-41FB-9F29-1AB718874128	#idACD1B0A9-F870-4B46-88CF-C870A9EDAF8B	93	Restricted Local Access Road	Single Carriageway
4	#id03312900-B147-4CA3-A858-E2BF6AD1ECA7	#id02FE73D4-E88D-4119-8DC2-6E80DE6F6594	95	Restricted Local Access Road	Single Carriageway

In [7]:
```
road_edges.dtypes
```

Out[7]:
```
src_id     object
dst_id     object
length      int64
type       object
form       object
dtype: object
```

In [8]:
```
road_edges.shape
```

Out[8]:
```
(3725531, 5)
```

In [9]:
```
road_edges['type'].unique()
```

Out[9]:
```
0                          A Road
1                          B Road
2               Local Access Road
3                      Local Road
4                      Minor Road
5                        Motorway
6    Restricted Local Access Road
7           Secondary Access Road
Name: type, dtype: object
```

In [10]:
```
road_edges['form'].unique()
```

Out[10]:
```
0    Collapsed Dual Carriageway
1              Dual Carriageway
2                 Guided Busway
3                    Roundabout
4        Shared Use Carriageway
5            Single Carriageway
6                     Slip Road
Name: form, dtype: object
```

Exercise: Make IDs Compatible

Our csv files were derived from original GML files, and as you can see from the above, both `road_edges['src_id']` and `road_edges['dst_id']` contain a leading `#` character that `road_nodes['node_id']` does not. To make the IDs compatible between the edges and nodes, use cuDF's string method `.str.lstrip` to replace the `src_id` and `dst_id` columns in `road_edges` with values stripped of the leading `#` characters.

In [12]:
```
road_edges['src_id'] = road_edges['src_id'].str.lstrip('#')
road_edges['dst_id'] = road_edges['dst_id'].str.lstrip('#')
road_edges[['src_id', 'dst_id']].head()
```

Out[12]:

	src_id	dst_id
0	id138447A5-91D4-4642-BFAC-13F309705429	id84C9DAD4-9243-4742-B582-E8CBC848E08A
1	idD615F9C5-5BE9-412D-9FED-F4928BAB4146	idA1BB20B9-0751-4B42-9925-20607ABF5027
2	idDC14D4D1-774E-487D-8EDE-60B129E5482C	id51555819-1A39-4B41-B0C9-C6D2086D9921
3	id626FC567-199C-41FB-9F29-1AB718874128	idACD1B0A9-F870-4B46-88CF-C870A9EDAF8B
4	id03312900-B147-4CA3-A858-E2BF6AD1ECA7	id02FE73D4-E88D-4119-8DC2-6E80DE6F6594

Solution

In [13]:
```
# %load solutions/make_ids_compatible
road_edges['src_id'] = road_edges['src_id'].str.lstrip('#')
road_edges['dst_id'] = road_edges['dst_id'].str.lstrip('#')
road_edges[['src_id', 'dst_id']].head()
```

Out[13]:

	src_id	dst_id
0	id138447A5-91D4-4642-BFAC-13F309705429	id84C9DAD4-9243-4742-B582-E8CBC848E08A
1	idD615F9C5-5BE9-412D-9FED-F4928BAB4146	idA1BB20B9-0751-4B42-9925-20607ABF5027
2	idDC14D4D1-774E-487D-8EDE-60B129E5482C	id51555819-1A39-4B41-B0C9-C6D2086D9921
3	id626FC567-199C-41FB-9F29-1AB718874128	idACD1B0A9-F870-4B46-88CF-C870A9EDAF8B
4	id03312900-B147-4CA3-A858-E2BF6AD1ECA7	id02FE73D4-E88D-4119-8DC2-6E80DE6F6594

Data Summary

Now that the data is cleaned we can see just how many roads and endpoints/junctions/curve points we will be working with, as well as its memory footprint in our GPU. The GPUs we are using can hold and analyze much larger graphs than this one!

In [14]:
```
print(f'{road_edges.shape[0]} edges, {road_nodes.shape[0]} nodes')
```

```
3725531 edges, 3121148 nodes
```

In [15]:
```
!nvidia-smi
```

```
Thu Nov 16 00:52:50 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:1B.0 Off |                    0 |
| N/A   28C    P0    25W /  70W |   6219MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            On   | 00000000:00:1C.0 Off |                    0 |
| N/A   22C    P8     8W /  70W |      3MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla T4            On   | 00000000:00:1D.0 Off |                    0 |
| N/A   22C    P8     9W /  70W |      3MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla T4            On   | 00000000:00:1E.0 Off |                    0 |
| N/A   22C    P8     8W /  70W |      3MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Building the Road Network Graph

We don't have information on the direction of the roads (some of them are one-way), so we will assume all of them are two-way for simplicity. That makes the graph "undirected," so we will build a cuGraph `Graph` rather than a directed graph or `DiGraph` .

We initialize it with edge sources, destinations, and attributes, which for our data will be the length of the roads:

In [16]:
```
G = cg.Graph()
%time G.from_cudf_edgelist(road_edges, source='src_id', destination='dst_id', edge_att
```

```
CPU times: user 304 ms, sys: 161 ms, total: 465 ms
Wall time: 463 ms
```

Just as a point of comparison, we also construct the equivalent graph in NetworkX from the equivalent cleaned and prepped Pandas dataframe.

In [17]:
```
road_edges_cpu = road_edges.to_pandas()
%time G_cpu = nx.convert_matrix.from_pandas_edgelist(road_edges_cpu, source='src_id',
```

```
CPU times: user 16 s, sys: 1.15 s, total: 17.1 s
Wall time: 17.1 s
```

Reindex `road_nodes`

For efficient lookup later, we will reindex `road_nodes` to use the `node_id` as its index - remember, we will typically get results from the graph analytics in terms of `node_id` s, so this lets us easily pull other information about the nodes (like their locations). We then sort the dataframe on this new index:

In [18]:
```
road_nodes = road_nodes.set_index('node_id', drop=True)
%time road_nodes = road_nodes.sort_index()
road_nodes.head()
```

```
CPU times: user 262 ms, sys: 11.8 ms, total: 274 ms
Wall time: 274 ms
```

Out[18]:

node_id	east	north	type
id000000F5-5180-4C03-B05D-B01352C54F89	432920.250	572547.4375	road end
id000003F8-9E09-4829-AD87-6DA4438D22D8	526616.375	189678.3906	junction
id000010DA-C89A-4198-847A-6E62815E038A	336879.000	731824.0000	junction
id000017A0-1843-4BC7-BCF7-C943B6780839	380635.000	390153.0000	junction
id00001B2A-155F-4CD3-8E06-7677ADC6AF74	337481.000	350509.7188	junction

Analyzing the Graph

Now that we have created the graph we can analyze the number of nodes and edges in it:

In [19]:
```
G.number_of_nodes()
```

Out[19]:
```
3078117
```

In [20]:
```
G.number_of_edges()
```

Out[20]:
```
3620793
```

Notice that the number of edges is slightly smaller than the number of edges in `road_edges` printed above--the original data came from map tiles, and roads that passed over the edge of a tile were listed in both tiles, so cuGraph de-duplicated them. If we were creating a `MultiGraph` or `MultiDiGraph` --a graph that can have multiple edges in the same direction between nodes--then duplicates could be preserved.

We can also analyze the degrees of our graph nodes:

In [21]:
```
deg_df = G.degree()
```

In an undirected graph, every edge entering a node is simultaneously an edge leaving the node, so we expect the nodes to have a minimum degree of 2:

In [22]:
```
deg_df['degree'].describe()[1:]
```

Out[22]:
```
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
```

You will spend more time using this GPU-accelerated graph later in the workshop.

Exercise: Construct a Graph of Roads with Time Weights

For this series of exercises, you are going to construct and analyze a new graph of Great Britain's roads using the techniques just demonstrated, but this time, instead of using raw distance for the edges' weights, you will be using the time it will take to travel between the two nodes at a notional speed limit.

You will be beginning this exercise with the `road_edges` dataframe from earlier:

In [23]:
```
road_edges.dtypes
```

Out[23]:
```
src_id    object
dst_id    object
length     int64
type      object
form      object
dtype: object
```

Road Type to Speed Conversion

In order to calculate how long it should take to travel along a road, we need to know its speed limit. We will do this by utilizing `road_edges['type']` , along with rules for the speed limits for each type of road.

Here are the unique types of roads in our data:

In [24]:
```
road_edges['type'].unique()
```

Out[24]:
```
0                          A Road
1                          B Road
2               Local Access Road
3                      Local Road
4                      Minor Road
5                        Motorway
6    Restricted Local Access Road
7           Secondary Access Road
Name: type, dtype: object
```

And here is a table with assumptions about speed limits we can use for our conversion:

In [25]:
```
# https://www.rac.co.uk/drive/advice/legal/speed-limits/
# Technically, speed limits depend on whether a road is in a built-up area and the for
# but we can use road type as a proxy for built-up areas.
# Values are in mph.

speed_limits = {'Motorway': 70,
               'A Road': 60,
               'B Road': 60,
               'Local Road': 30,
               'Local Access Road': 30,
               'Minor Road': 30,
               'Restricted Local Access Road': 30,
               'Secondary Access Road': 30}
```

We begin by creating `speed_gdf` to store each road type with its speed limit:

In [26]:
```
speed_gdf = cudf.DataFrame()

speed_gdf['type'] = speed_limits.keys()
speed_gdf['limit_mph'] = [speed_limits[key] for key in speed_limits.keys()]
speed_gdf
```

Out[26]:

	type	limit_mph
0	Motorway	70
1	A Road	60
2	B Road	60
3	Local Road	30
4	Local Access Road	30
5	Minor Road	30
6	Restricted Local Access Road	30
7	Secondary Access Road	30

Next we add an additional column, `limit_m/s` , which for each road type will give us a measure of how fast one can travel on it in meters / second.

In [27]:
```
# We will have road distances in meters (m), so to get road distances in seconds (s),
# 1 mile ~ 1609.34 m
speed_gdf['limit_m/s'] = speed_gdf['limit_mph'] * 1609.34 / 3600
speed_gdf
```

Out[27]:

	type	limit_mph	limit_m/s
0	Motorway	70	31.292722
1	A Road	60	26.822333
2	B Road	60	26.822333
3	Local Road	30	13.411167
4	Local Access Road	30	13.411167
5	Minor Road	30	13.411167
6	Restricted Local Access Road	30	13.411167
7	Secondary Access Road	30	13.411167

Step 1: Merge `speed_gdf` into `road_edges`

cuDF provides merging functionality just like Pandas. Since we will be using values in `road_edges` to construct our graph, we need to merge `speed_gdf` into `road_edges` (similar to a database join). You can merge on the `type` column, which both of these dataframes share.

In [28]:
```
%time road_edges = road_edges.merge(speed_gdf, on='type')
```

```
CPU times: user 29.6 ms, sys: 4.38 ms, total: 33.9 ms
Wall time: 33.1 ms
```

Step 2: Add Length in Seconds Column

You now need to calculate the number of seconds it will take to traverse a given road at the speed limit. This can be done by dividing a road's length in m by its speed limit in m/s. Perform this calculation on `road_edges` and store the results in a new column `length_s`.

In [30]:
```
road_edges['length_s'] = road_edges['length'] / road_edges['limit_m/s']
road_edges['length_s'].head()
```

Out[30]:
```
0    3.280848
1    5.294096
2    5.219531
3    3.206283
4    5.144966
Name: length_s, dtype: float64
```

Solution

In [31]:
```
# %load solutions/length_in_seconds
road_edges['length_s'] = road_edges['length'] / road_edges['limit_m/s']
road_edges['length_s'].head()
```

Out[31]:
```
0    3.280848
1    5.294096
2    5.219531
3    3.206283
4    5.144966
Name: length_s, dtype: float64
```

Step 3: Construct the Graph

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

In [33]:
```
G_ex = cg.Graph()
G_ex.from_cudf_edgelist(road_edges, source='src_id', destination='dst_id', edge_attr='
```

Solution

In [34]:
```
# %load solutions/construct_graph
G_ex = cg.Graph()
G_ex.from_cudf_edgelist(road_edges, source='src_id', destination='dst_id', edge_attr='
```

Please Restart the Kernel

In [35]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[35]:
```
{'status': 'ok', 'restart': True}
```

Next

In the next notebook you will work with a data set representing a population 5 times larger than the UK, a data set that would not fit in the memory of a single GPU. In order to work with this data you will use Dask cuDF to partition the data among the 4 GPUs at your disposal, and perform the same kinds of data manipulations you have been doing with vanilla cuDF on smaller, single-GPU data sets.