# Dublin Road Speeds

Ruaridh Williamson

Supervised by Dessie Petrova

SATALIA

# Context

*Explore the historical travel speeds of Dublin's roads*

- **Shapefile** containing the geometry of road links located in Dublin. The speed for each road link is recorded at 15 minute intervals over one Thursday.
- **CSV** containing the road link attribute data
  - Road link length (metres)
  - Function class (road type)
  - Urban/Rural flag
  - Travel direction (True or False)
  - Road speed (km/h every 15min from 0:00 to 23:45)

# Agenda

- **Initial exploration**
    - Variable summaries and distributions
    - Confirmatory analysis
- **Understanding the dataset**
    - What caveats apply?
    - How does it look like the data has been measured and treated?
    - Variable creation
- **Use cases**
    - Visualising speeds over time
    - Traffic levels
    - Network algorithms
    - Commutability rating

# Initial Exploration

## Primary key

Start with CSV

What is the dataset *Point of View*?

- Road Link
- Travel Direction

...almost

| | Trav Dir | | |
|---|---|---|---|
| Link Id | False | True | G.. |
| 549454034 | 1 | 1 | 2 |
| 549454061 | 1 | 1 | 2 |
| 549454069 | 1 | 1 | 2 |
| 549454092 | 1 | 1 | 2 |
| 549454099 | 1 | 1 | 2 |
| 549454100 | 1 | 1 | 2 |
| 549454103 | 1 | 1 | 2 |
| 549454128 | 1 | 1 | 2 |
| 549454133 | 1 | 1 | 2 |
| 1143298779 | 1 | | 1 |
| 1143298780 | 1 | | 1 |
| 1143298781 | 1 | | 1 |
| 1143298782 | | 1 | 1 |
| 1143298783 | | 1 | 1 |
| 1143298784 | | 1 | 1 |
| 1143298785 | | 1 | 1 |
| 1143298786 | | 1 | 1 |

# Initial Exploration

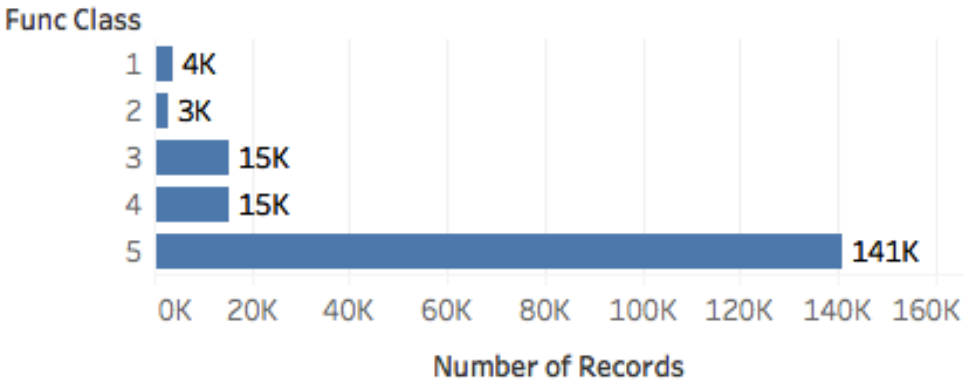
## Primary key

Start with CSV

What is the dataset *Point of View*?

- Road Link
- Travel Direction

…almost




| | Trav Dir | | |
|---|---|---|---|
| Link Id | False | True | G.. |
| 549454034 | 1 | 1 | 2 |
| 549454061 | 1 | 1 | 2 |
| 549454069 | 1 | 1 | 2 |
| 549454092 | 1 | 1 | 2 |
| 549454099 | 1 | 1 | 2 |
| 549454100 | 1 | 1 | 2 |
| 549454103 | 1 | 1 | 2 |
| 549454128 | 1 | 1 | 2 |
| 549454133 | 1 | 1 | 2 |
| 1143298779 | 1 | | 1 |
| 1143298780 | 1 | | 1 |
| 1143298781 | 1 | | 1 |
| 1143298782 | | 1 | 1 |
| 1143298783 | | 1 | 1 |
| 1143298784 | | 1 | 1 |
| 1143298785 | | 1 | 1 |
| 1143298786 | | 1 | 1 |

# Initial Exploration

## Tidy data

Convert from *wide* to *long* format

```
# Melt data columns labelled as u00_00 ... u23_45 into one variable "Time" with value "Speed"
melted_dt <- data.table::melt(dublin_csv, measure.vars = grep("u\\d", names(dublin_csv), value = TRUE),
                              variable.name = "Time", value.name = "Speed")

# Convert Time variable into datetime representation
melted_dt <- melted_dt[, Time := as.POSIXct(Time, format = "u%H_%M")]
```

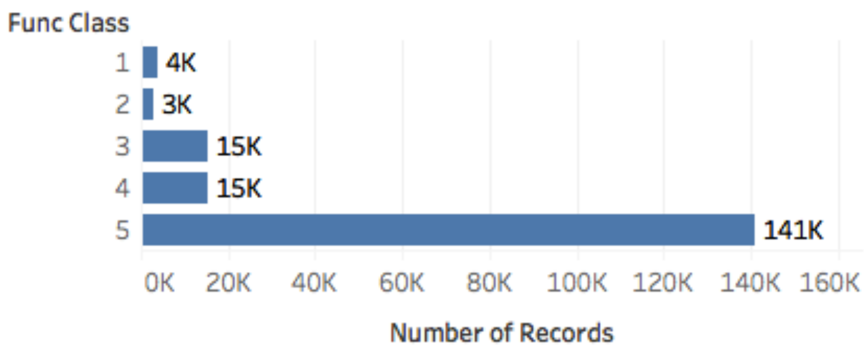
PoV is now Link (100k) by Time (96) by Direction (2)

Nearly 20m rows, ~1.5GB on disk

# Initial Exploration

## What does Function Class represent?

### Distribution of Function Class (indicates the road category)




| | Func Class 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Avg. U00 00 | 64 | 46 | 45 | 42 | 24 |
| Avg. U00 15 | 63 | 46 | 45 | 42 | 24 |
| Avg. U00 30 | 63 | 46 | 45 | 42 | 24 |
| Avg. U00 45 | 63 | 46 | 45 | 42 | 24 |
| Avg. U01 00 | 63 | 46 | 45 | 42 | 24 |
| Avg. U01 15 | 63 | 46 | 45 | 42 | 24 |
| Avg. U01 30 | 63 | 46 | 45 | 42 | 24 |

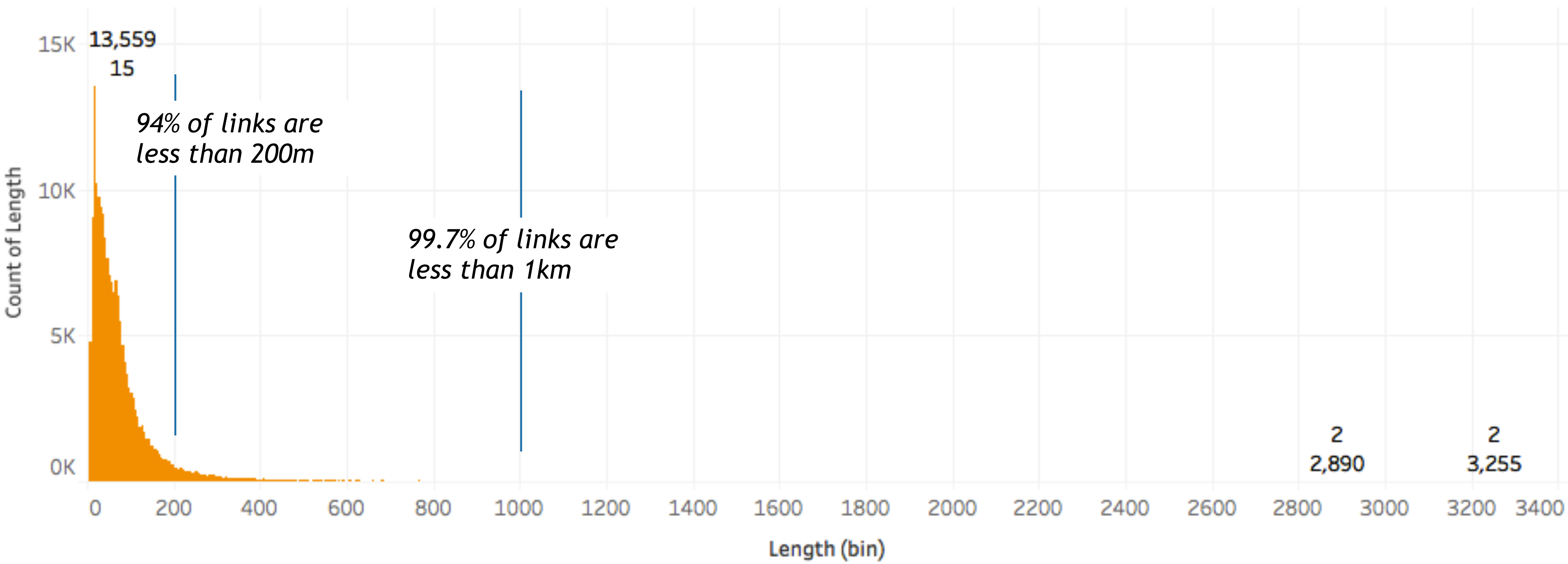
*Average Speed (km/h) by time and Func Class*

… so 1 is a highway and 5 is suburban

# Initial Exploration

How are the road lengths distributed?

## Distribution of Road Lengths

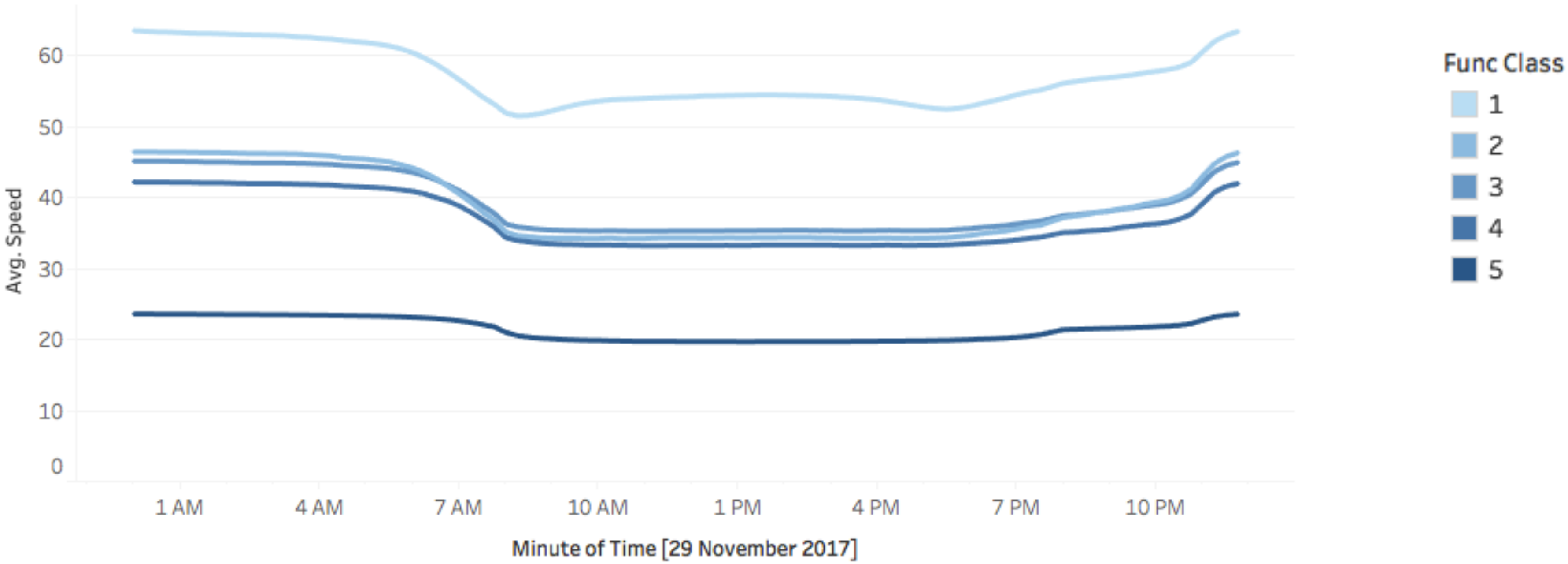



*Road Lengths (metres) grouped by 5m bins*

# Initial Exploration

How are the road speeds distributed?

## Distribution of Speed over time



# Understanding the Dataset

Grouping 15min intervals throughout the day

3rd party pre-processing

# Dataset Treatment

Clustering times of the day for collective analysis

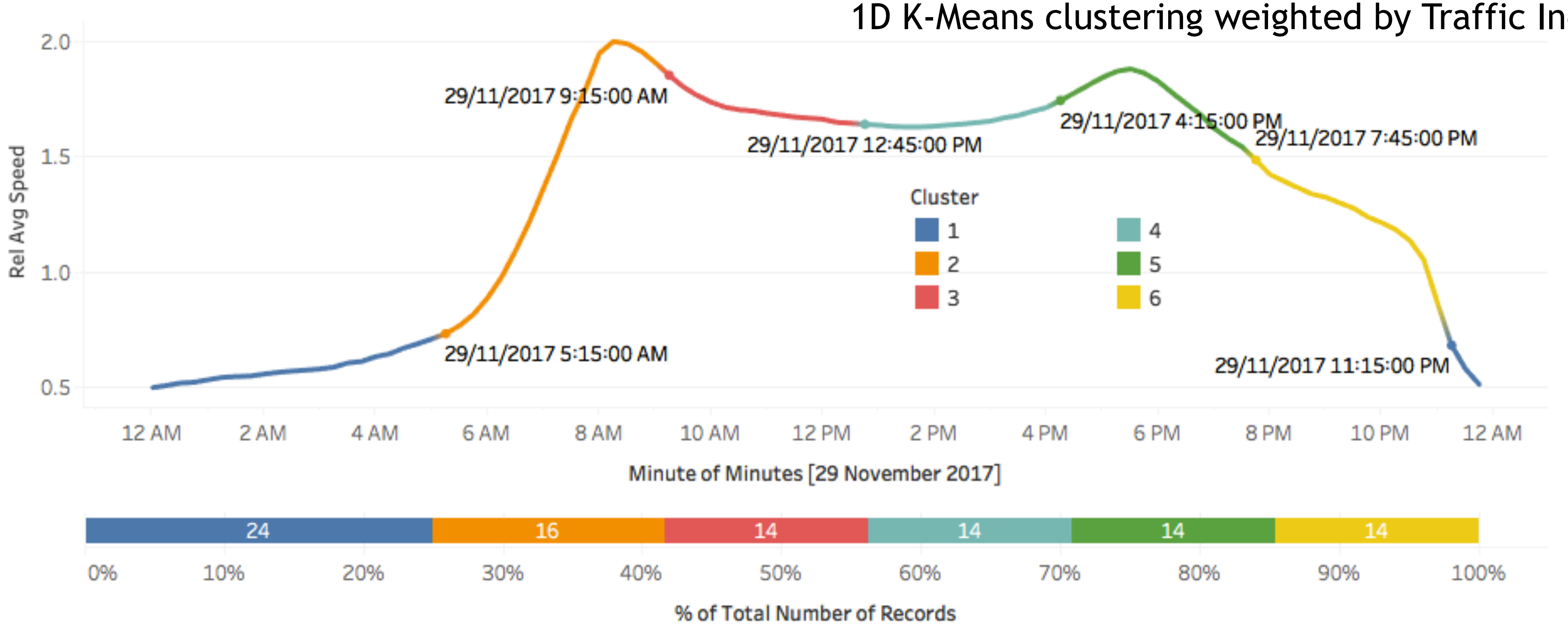
## Traffic Index over time by Cluster

# Dataset Treatment

Clustering times of the day for collective analysis

## Traffic Index over time by Cluster

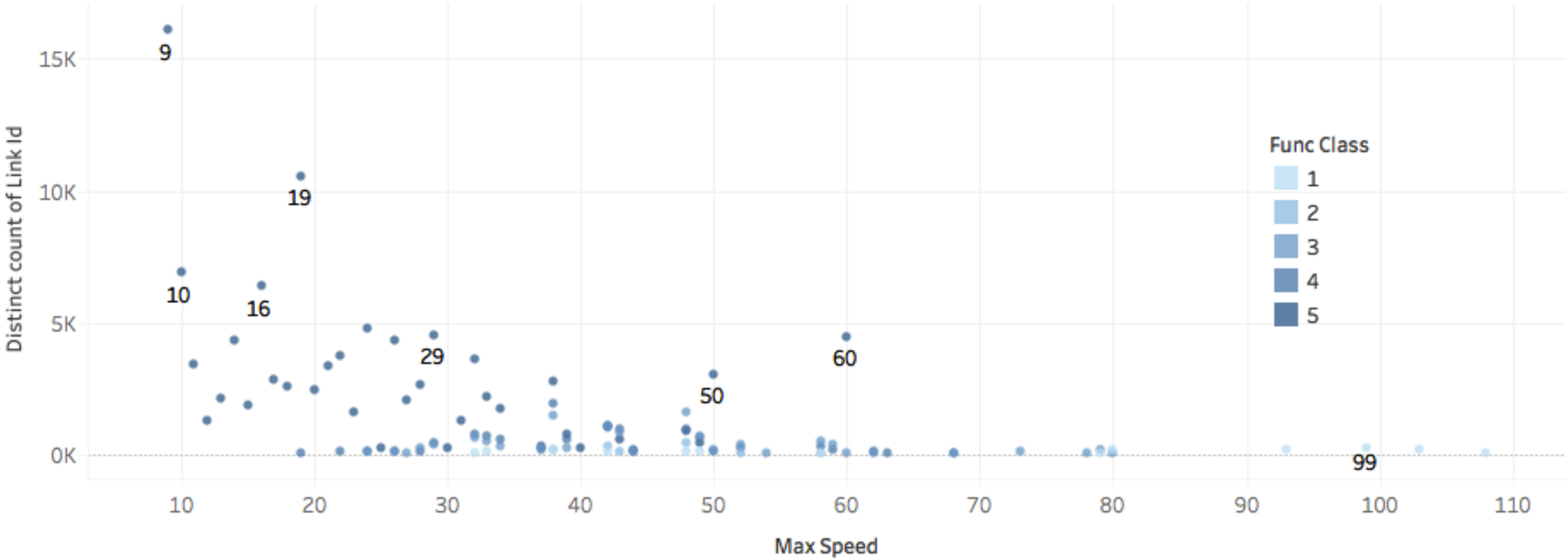
1D K-Means clustering weighted by Traffic Index



# Dataset Treatment

Would we expect huge counts of rounded numbers?

## Distribution of Max Speed (a continuous variable with no binning...)

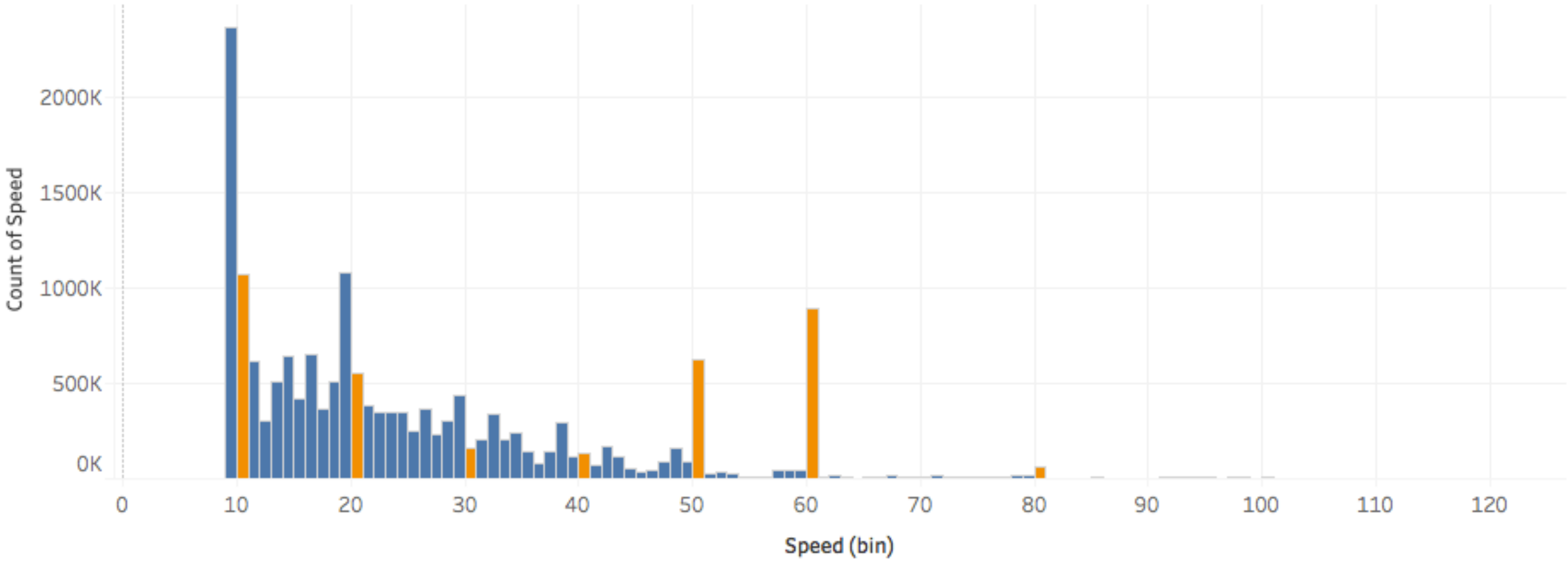



*Speeds where the count < 100 have been hidden*

# Dataset Treatment

Speed intervals of 10 highlighted

## Distribution of Speed (a continuous variable with no binning…)

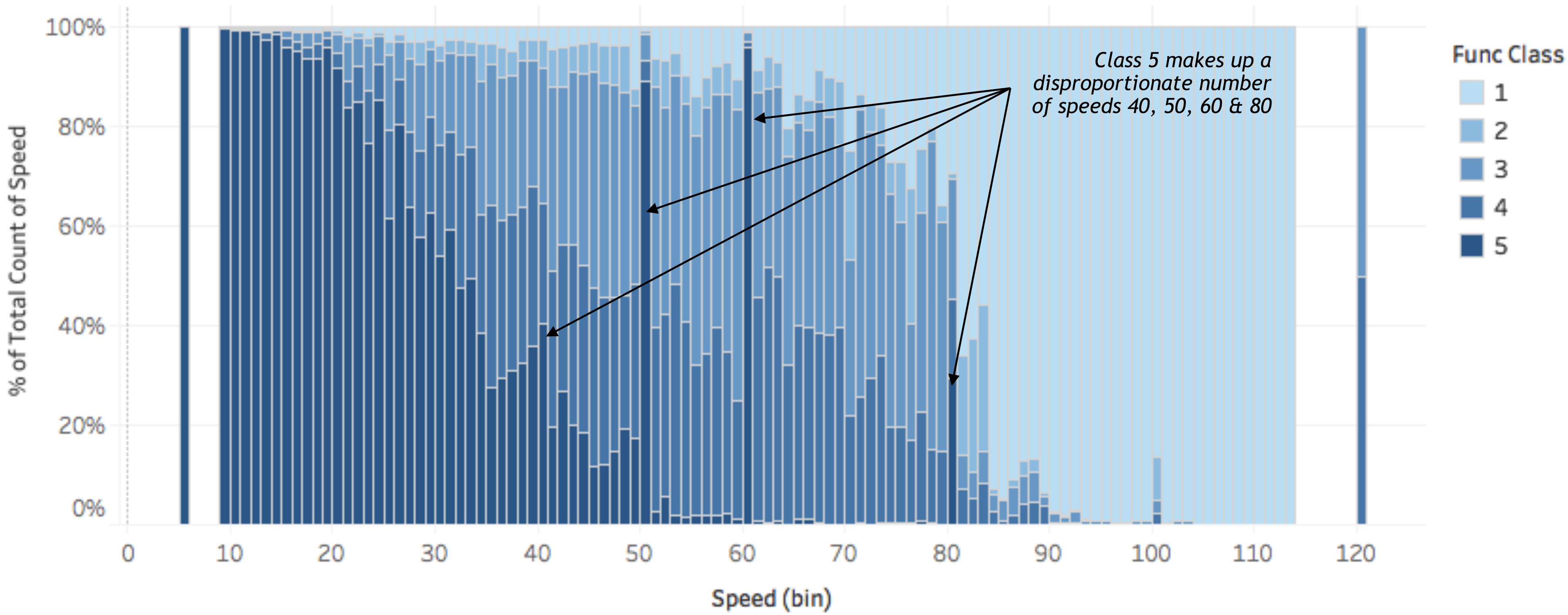



*Bin intervals are 1km/h*

# Dataset Treatment

Proportion of Func Class making up each Speed value

## Distribution of Speed (a continuous variable with no binning…)




*Bin intervals are 1km/h*

# Use Cases

Grass GIS

Python NetworkX

Spatio-temporal visualisation

# Use Cases

## GRASS GIS

### Build a connected graph object from the Shapefiles

```
v.net --verbose input=dub30_exp_thu points=nodes out=streets_net operation=connect threshold=10
```

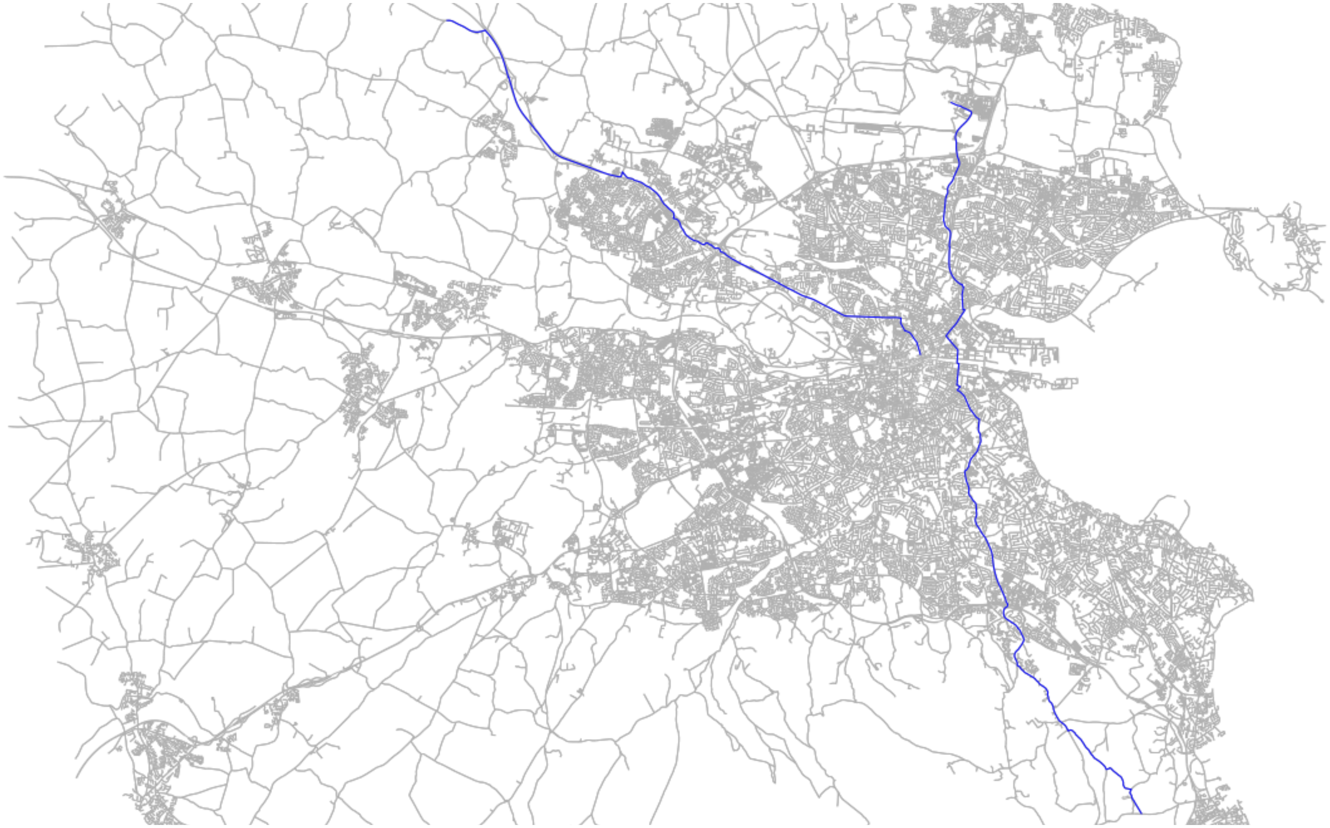
### Compute Shortest Path between two points

```
v.net.path input=streets_net output=path arc_column=length
```

# Use Cases

## GRASS GIS Shortest Path

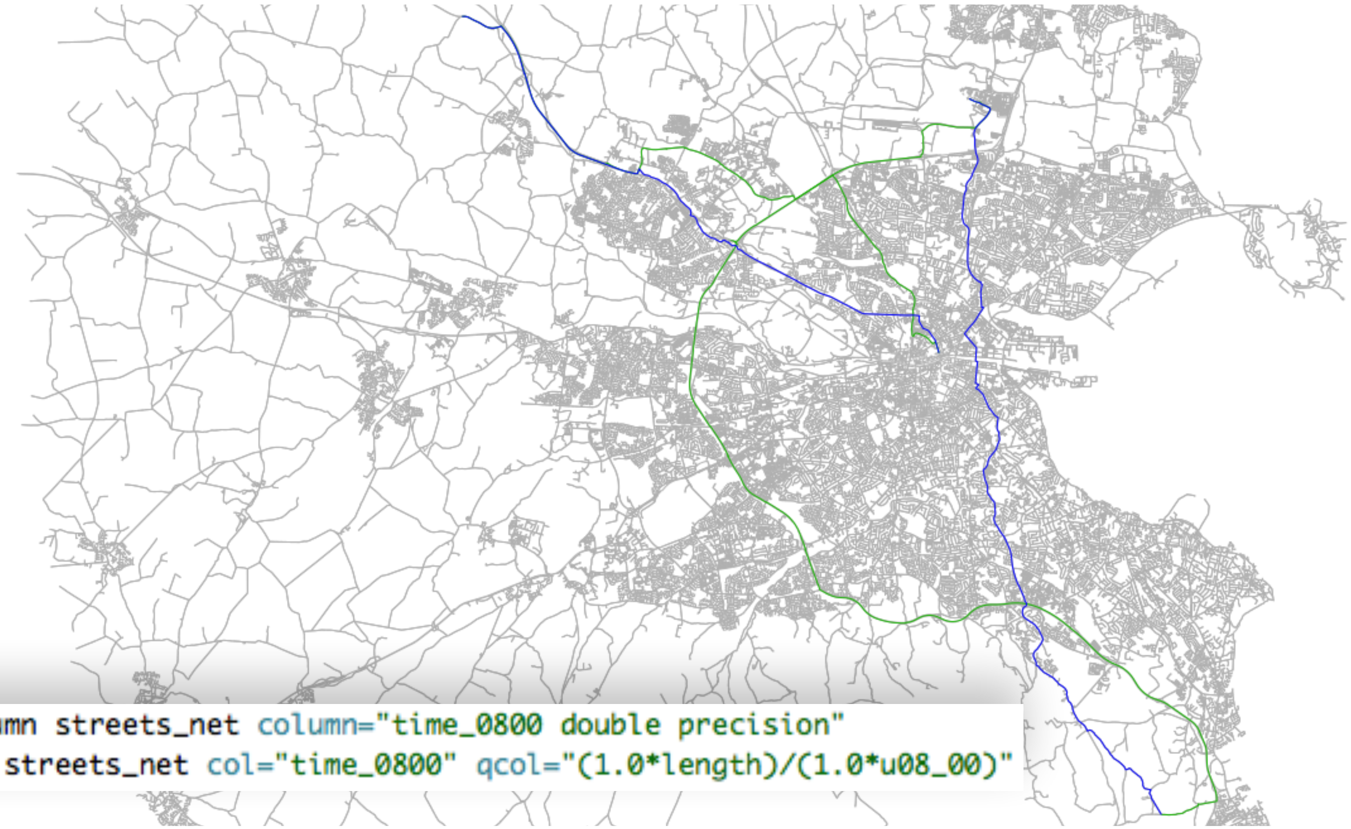
Compute Shortest Path between two points *by Length*



# Use Cases

## GRASS GIS Shortest Path

### Compute Shortest Path between two points *by Time*



```
v.db.addcolumn streets_net column="time_0800 double precision"
v.db.update streets_net col="time_0800" qcol="(1.0*length)/(1.0*u08_00)"
```

*Shortest Path at 8:00*

# Use Cases

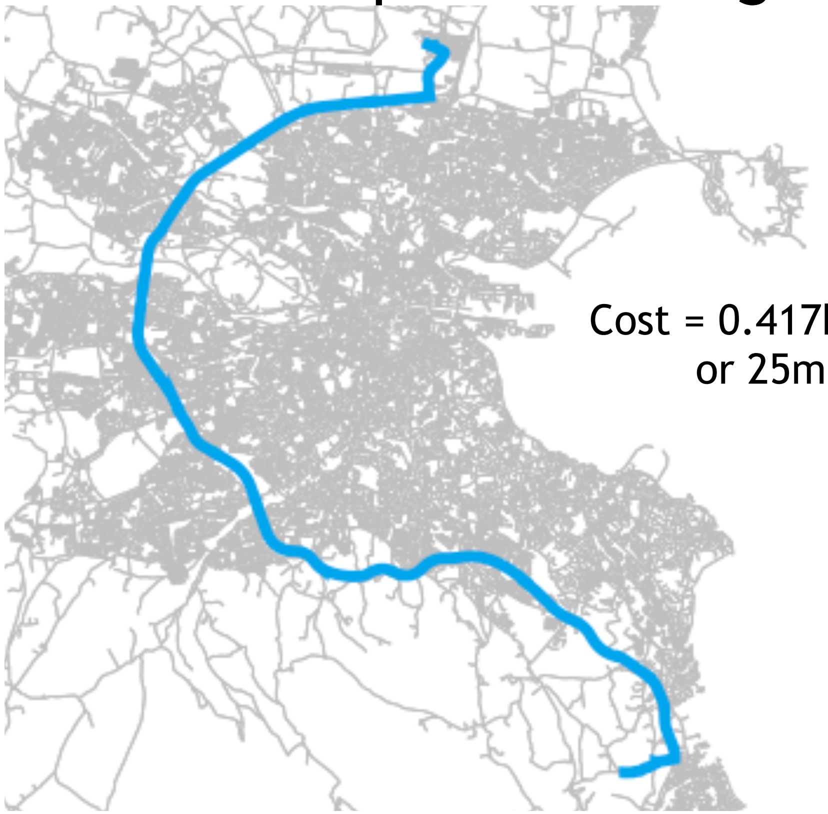
## GRASS GIS Shortest Path

Compute Shortest Path between two points *by Speed*

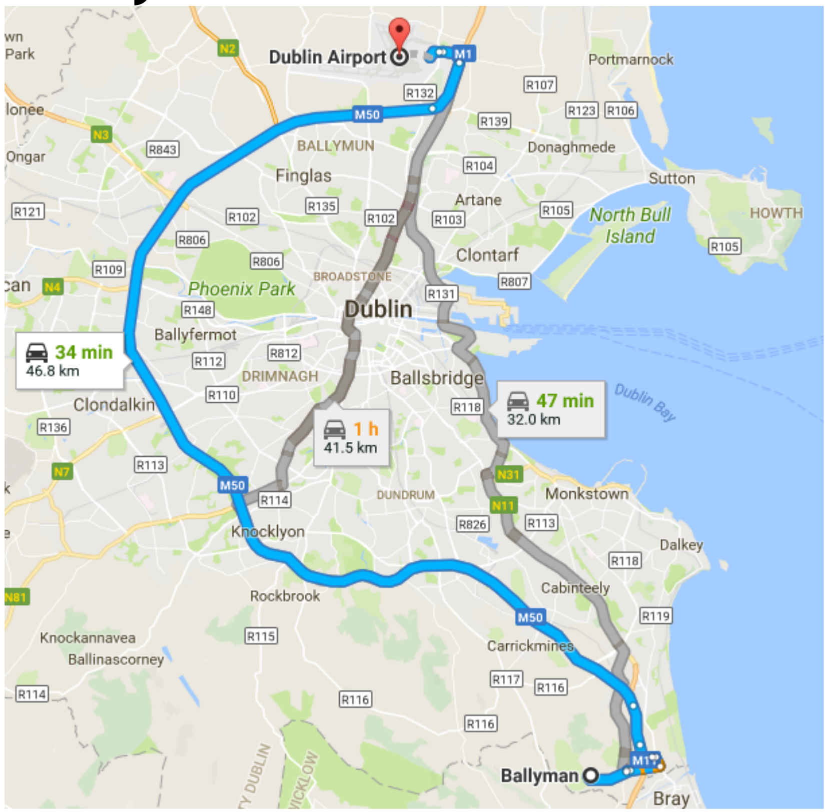
# Use Cases

## GRASS GIS Shortest Path

### What do the experts at Google Maps say?



Cost = 0.417hrs
or 25mins



# Use Cases

## Python's NetworkX

Use a dedicated Graph analysis package…

# Dublin Networkx

## Analysing Dublin’s road network using NetworkX

This Notebook provides an overview of importing the Shapefile data into the NetworkX Python library

## Exploration

Begin by importing the modules and required GRASS data

In [1]:
```
import networkx as nx # use patched fork from github.com/ruaridhw/networkx/tree/wr
ite-shp-dev
%matplotlib inline
```

In [2]:
```
%time G = nx.read_shp('../2_grass_data_output/connections_shp/connections.shp')
```

```
CPU times: user 2min 45s, sys: 1.06 s, total: 2min 46s
Wall time: 2min 47s
```

Examine the number of nodes and edges:

In [3]:
```
print(nx.info(G))
```

```
Name:
Type: DiGraph
Number of nodes: 85119
Number of edges: 95867
Average in degree:   1.1263
Average out degree:   1.1263
```

What is the degree of each node?

In [4]:
```
nx.degree_histogram(G)
```

Out[4]:
```
[0, 21221, 24358, 36432, 3043, 61, 4]
```

Is the road network strongly connected as expected?

In [5]:
```
nx.is_strongly_connected(G)
```

Out[5]:
```
False
```

In [6]:
```
nx.number_weakly_connected_components(G)
```

Out[6]:
```
36
```

Print out the first 10 edges:

In [7]:
```
for i, edge in enumerate(G.edges()):
    if i == 10:
        break
    print(edge)
```

```
((317995.5708697437, 245878.08877053304), (317973.5650240276, 245907.605849580
08))
((317995.5708697437, 245878.08877053304), (318026.7365888019, 245907.816629536
94))
((317973.5650240276, 245907.60584958008), (318382.4449278968, 246117.125933518
14))
((317960.94469078025, 245853.84258207353), (317941.54076457256, 245885.6516297
2905))
((317960.94469078025, 245853.84258207353), (317986.6586180811, 245862.27751578
222))
((317941.54076457256, 245885.65162972905), (317942.0665525985, 245891.23232511
77))
((317894.5420938695, 245824.35177009527), (317960.94469078025, 245853.84258207
353))
((317894.5420938695, 245824.35177009527), (317941.54076457256, 245885.65162972
905))
((317942.0665525985, 245891.2323251177), (317932.9315142884, 245937.7728218606
8))
((317942.0665525985, 245891.2323251177), (317952.4183059603, 245902.6252452652
6))
```

Print out the node attributes:

In [8]:
```
print(G.nodes[(317995.5708697437, 245878.08877053304)])
```

```
{}
```

No nodes have *any* attributes

In [9]:
```
for i, node in enumerate(G.nodes()):
    if len(G.nodes[node]) > 0:
        print(node)
```

What does the edge data look like?

In [10]:
```
G.edges[(317995.5708697437, 245878.08877053304), (317973.5650240276, 245907.60584958008)]
```

Out[10]:
```
{'Json': '{ "type": "LineString", "coordinates": [

                                                  ] }',
 'ShpName': 'connections',
 'Wkb': b'

 ',
 'Wkt': 'LINESTRING


 'cat': 1,
 'func_class': 3,
 'length': 40,
 'link_id':


 'trav_dir': 'T',
 'u00_00': 48,
 'u00_15': 48,
```

Print out the 8am speed for the first 10 edges:

In [11]:
```
for i, (edge, speed) in enumerate(nx.get_edge_attributes(G,'u08_00').items()):
    if i == 10:
        break
    print(speed)
```

```
34
33
54
30
27
39
38
33
36
35
```

What does the graph structure of the first Weakly Connected Component look like?

In [12]:

```
for i, graph in enumerate(nx.weakly_connected_component_subgraphs(G)):
    print(nx.info(graph))
    if i == 0:
        break
```

```
Name:
Type: DiGraph
Number of nodes: 84918
Number of edges: 95694
Average in degree:   1.1269
Average out degree:   1.1269
```

## Calculated Attributes

Calculate the travel time (in hours) for every road at 5:15pm

In [13]:

```
d = {}
for (n1, n2) in G.edges():
    e = G[n1][n2]
    d[(n1, n2)] = (e['length'] / 1000.0) / (e['u17_15'] * 1.0) # Convert 'length'
 from metres to km
nx.set_edge_attributes(G, d, 'time_1715')
```

In [14]:

```
print('Length (m): ', G[n1][n2]['length'])
print('Speed (km/h): ', G[n1][n2]['u17_15'])
print('Time (secs): ', G[n1][n2]['time_1715'] * 60 * 60)
```

```
Length (m):  12
Speed (km/h):  17
Time (secs):  2.5411764705882356
```

# Network Algorithms

Calculate the **Edge Betweeness Centrality** for each edge by sampling 1000 other edges weighted by travel time.

> *For a given edge $e$, it is the sum over all node pairs of the fraction of all-pairs shortest paths that pass through $e$*

In [15]:
```
%time bc = nx.edge_betweenness_centrality(G, k=1000, normalized=False, weight='tim
e_1715', seed=999)
len(bc)
```

```
CPU times: user 4min 12s, sys: 1.79 s, total: 4min 14s
Wall time: 4min 13s
```

Out[15]:
```
95867
```

Calculate the **Page Rank** for each node with edges weighted by travel time.

> *A ranking of the nodes in the graph based on the structure of the incoming links.*

In [16]:
```
%time pr = nx.pagerank(G, weight = 'time_1715')
pr_dict = {node: 0 for node in G.nodes} # Create dummy dictionary with all nodes a
s zeroes
pr_dict.update(pr)
```

```
CPU times: user 6.2 s, sys: 1.28 s, total: 7.49 s
Wall time: 7.75 s
```

Calculate the **Shortest Path Lengths** from Dublin's *The Spire* tourist attraction to every other point in Dublin.

In [17]:
```
# Find geographical coordinates of The Spire node as a tuple
approx_spire = (315920.300778, 234593.104858) # Estimate Coordinates from GRASS
for node in G.nodes():
    x, y = node
    if abs(approx_spire[0] - x) + abs(approx_spire[1] - y) < 1:
        spire = node # Find nearest node to within Manhattan Distance of 1
        break
```

In [18]:
```
%time path_lengths = nx.shortest_path_length(G, source = spire, target = None, wei
ght = 'time_1715')
```

```
CPU times: user 31.5 ms, sys: 26 ms, total: 57.5 ms
Wall time: 56.3 ms
```

If the graph were strongly connected we would expect every node to be reachable from *The Spire*...

In [19]:
```
pl_dict = {node: 0 for node in G.nodes} # Create dummy dictionary with all nodes a
s zeroes
pl_dict.update(path_lengths)
len(path_lengths)
```

Out[19]:
```
6086
```

Instead, only 6086 of the 85119 nodes are reachable

## Update Graph

Save the new metrics to graph structure.

In [20]:

```
nx.set_node_attributes(G, pl_dict, 'spire_time')
nx.set_node_attributes(G, pr_dict, 'page_rank')
nx.set_edge_attributes(G, bc, 'b_c')
```

## Output to .shp

Output our new metrics to a directory of shapefiles for further geospatial analysis

In [21]:
```
nx.write_shp(G, '../6_python_data_output/networkx_shp')
```

## Visualising the Graph

Use a dedicated Graph visualisation package…

In [24]:
```
# Get a subgraph of 100 nodes
H = G.subgraph([node for i, node in enumerate(G.nodes()) if i < 100])
```

Push subgraph to a local Docker Neo4j graph database

In [25]:
```
import neonx # from github.com/ruaridhw/neonx/tree/labels-auth
graph_db = 'http://localhost:7474/db/data/'
login = 'neo4j'
pw = 'focused_leavitt'
relationship = 'ROAD_TO'

results = neonx.write_to_neo(graph_db, H, relationship, server_login = login, serv
er_pwd = pw)
```

Directed graph of 100 random nodes with edge road speed data preserved.

Each node is coloured using the minimum Function Class of the incident edges