Task A (spatial networks and planarity)

In order to select an area of Leeds with a high number of accidents, initial research was conducted by consulting a map of the population density of Leeds [1]. The analysis revealed that one of the areas with the highest population density was located in Leeds town centre, situated above the Leeds train station. It is noteworthy that Leeds station is the most utilized train station in the district of Leeds [2] [3], suggesting that an area in close proximity to the station would likely have a higher frequency of road accidents.

A change of one kilometre equals a change of roughly 0.0111 latitude/longitude units [9] (although this is slightly different depending on the distance of these coordinates from the equator). As a result, as I was aiming for an area of roughly 1 square KM, I would want a change of around 0.01 latitude/longitude units between my four coordinates. The latitude/longitude values are declared as constants at the start of the code so that they can be easily changed to a different area of Leeds if needed. The coordinates I ended up using are displayed in Table 1 – the selection process is discussed in Task B's section. The program calculates the total area mapped by these coordinates to be 1.2343 square KM. The area mapped can be seen in Figure 1. An interactable map can be used to view this area using the footnote[1].

North	53.804
South	53.794
East	-1.536
West	-1.546

Table 1: Latitude /longitude of Leeds Town Centre area

Using the OSMnx Python package with these coordinates, the *generate_city_graph* function downloads the OpenStreetMap data of the specified area, solely for drivable roads, whereafter the *generate_roads* function fills in any missing data for any streets that were incorrectly imported. Following this, the *print_characteristics* function displays various statistics about the OpenStreetMap to the user. This includes but is not limited to the average circuitry of a network, which is defined by the average sum of the actual distances between edges divided by the average sum of direct distances between edges [4], as well as if the network is planar. These statistics can be seen in Table 2.

Number of unique accidents	337
Number of intersections	145
Number of Roads	159
Average number of roads per intersection	3.2956
Average number of streets per node	2.8994
Total length of streets	13800.5940
Average length of a street	63.0164
Average number of intersections per square KM	131.9501
Average number of streets per square KM	12558.5531
Spatial diameter	1910.6330
Sum of actual road lengths	16.9832
Sum of direct distances between nodes	13.2864
Average circuitry of network	1.0387

Table 2: Statistics about the Leeds Town Centre area

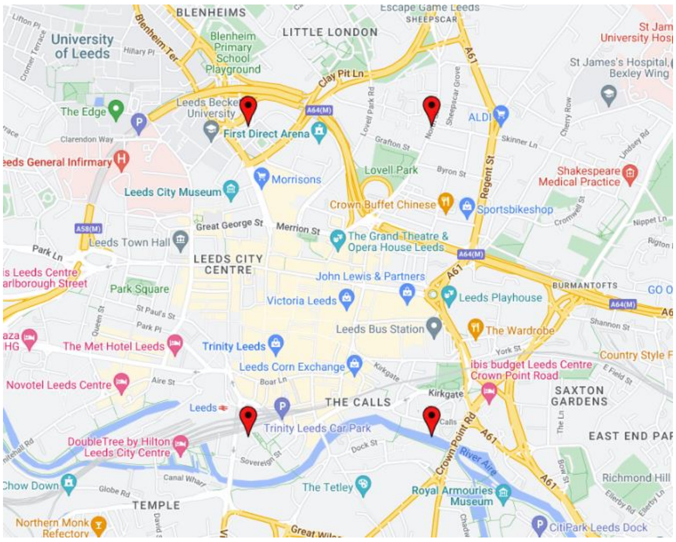

Figure 1: Leeds Town Centre area with plotted latitude/longitude coordinates (in Table 1) as red markers (An interactable map can be used to view this area using the footnote [1])

As seen in Table 2, the average circuitry of the network is 1.0387, which is quite small. A value of 1 would indicate that the roads are perfectly efficient, as each road between each intersection would be a straight line with no extra road length. A value of 1.0387, which is just slightly higher than 1, suggests that the road efficiency for the selected area is quite good, as the extra length of roads is minimal. This is helpful as this means that there is less traffic congestion and less fuel consumption by vehicles due to shorter travel times.

The function also calculates if the network is planar – for the area that I have described above, the network is indeed planar, meaning that it can be drawn without any of its edges, in this case, roads, crossing each other. This means that in my selected area, there are no complex road intersections or overlapping bridges (for motorists). This probably results in lower road maintenance costs, and no bridges or tunnels have to be maintained in this area.

[1] https://www.google.com/maps/d/u/0/edit?mid=10ethl34JwOmm3lMpSzWZlSlvDZnOO5k&usp=sharing

Task B (Road accidents)

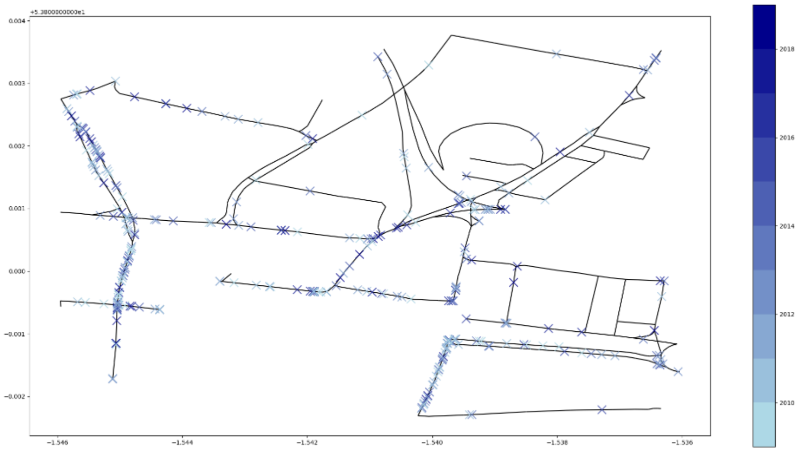

Figure 2: Selected Leeds Town Centre area with plotted car accidents 2009-2019

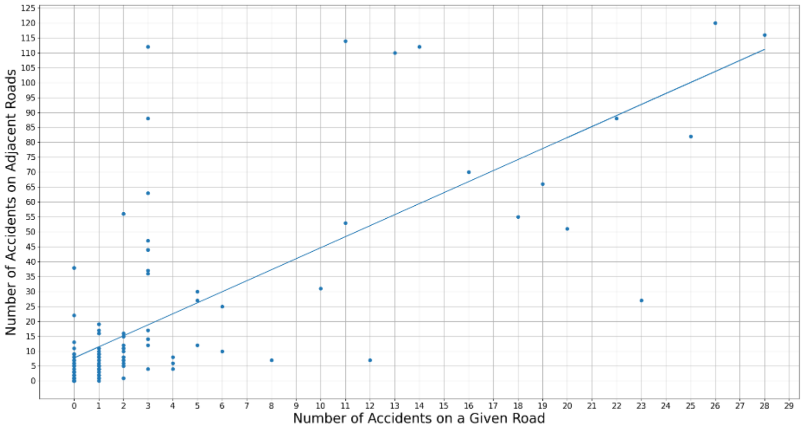

Figure 3: Scatter Plot of the number of accidents on a road compared to the number of accidents on adjacent roads

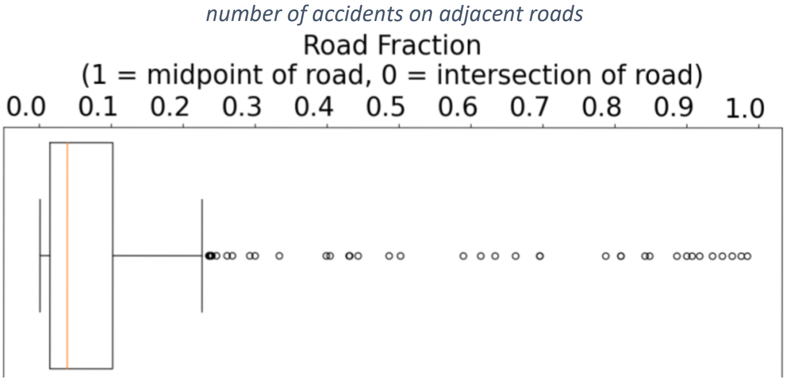

Figure 4: Boxplot of the distribution of accident road fractions, meaning how far down a road an accident happened, with 1 = midpoint of road, and 0 = intersection of road

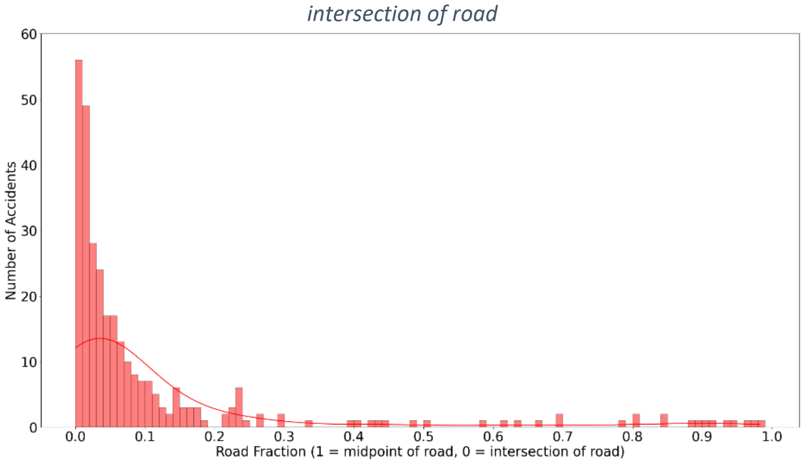

Figure 5: Histogram of the distribution of accident road fractions, meaning how far down a road an accident happened, with 1 = midpoint of road, and 0 = intersection of road

The process of selecting an appropriate area for analysis is a crucial step in conducting any spatial analysis. In this study, the area selection process involved downloading and plotting data onto a range of specified areas so as to identify a location with a large number of car accidents. To ensure that the selected area was suitable for analysis, the criteria for selection included having over 300 car accidents and being roughly 1 square KM in size.

To download the data for road accidents in Leeds, the *import_accidents* function was employed. This function compiles all the available accident data from 2009 to 2019 [10] (published by Leeds City Council) into one DataFrame and filters out accidents that did not occur within the specified area. The remaining accidents were then plotted onto the selected road network using the *print_map* function, depicted in Figure 2. The colour map in the figure is used to indicate the year of the accident, with darker shades of blue representing more recent accidents.

It is worth noting that multiple rows in the CSV files may reference the same accident, as each row denotes one casualty. As such, accidents resulting in multiple casualties span multiple rows in the CSV files. The program calculated a total of 337 unique car accidents recorded in the years 2009-2019 for the selected area.

To investigate the relationship between accidents on one road and those on connecting roads, several functions were written to collect and plot the relevant data. The *count_accidents* function calculated the total number of accidents on each road, while the *count_adjacent_accidents* function calculated the number of accidents that occurred on each of the adjacent roads connected to both intersection nodes of the said road. The counts were visualized in a scatter plot, shown in Figure 3, using the *plot_accidents* function. We can see a correlation between the number of accidents per road and the number of accidents on adjacent roads, suggesting a clustering effect around the most frequently used roads in the network.

The *investigate_intersections* function calculates, for each accident, how far down the road the accidents had happened, saving this as the "road_fraction" field in the accidents DataFrame. A value of 1 indicates that the accident happened at the exact midpoint of the road that it occurred on, while a value of 0 indicates that the accident happened at exactly that road's intersection. This data is plotted as a boxplot and histogram using the *plot_intersection_fractions* function, displayed in Figures 4 and 5. Figure 4 shows us that the largest majority of accidents occurred within the first 23% of a road following an intersection; most of the accidents that happen past this point are outliers. These findings highlight the importance of intersection design and suggest that targeted interventions around the most used intersections could potentially reduce the frequency of car accidents in the Leeds Town Centre area.

Task C (Voronoi diagrams)

Station (Voronoi cell)	Marathon Length (KM)
Leeds	41.874
Guiseley	41.994
Horsforth	42.244
New Pudsey	41.540
Garforth	42.399
Burley Park	42.326
Cross Gates	42.352
Morley	41.702
Woodlesford	42.108
Wigton Lane (Bus stop)	41.812

Table 3: Voronoi cells marathon track lengths

The *select_seeds* function selects 10 intersections as seed nodes using preselected train stations that are geographically dispersed across the city, thereby minimizing the distance that runners would need to travel to participate in the marathons. Train stations are good candidates for seed nodes as runners could either take the train from their closest stations or take the bus to their train stations, as train stations are usually well connected with other transport services in the city. I went down a list of the most popular train stations in Leeds [2] [3], choosing train stations that were evenly spread across the whole of Leeds. For example, I decided that both the Garforth and East Garforth train stations were located in close proximity to one another, and so decided to only use the Garforth station for this area. For each station, the latitude and longitude coordinates of the closest intersection to said train station were saved, whereafter the function calculated the closest seed node in my map of Leeds to each of these coordinates.

However, after selecting 9 train stations, it was observed that a significant portion of Northern Leeds did not have any train stations in the area. In order to ensure that residents in this area were also fairly included in the marathons, the Wigton Lane bus stop was identified as the final seed node. This ensured that all residents in the city had equal access to the marathon. The resulting seed nodes, highlighted in red in Figure 6, were evenly distributed across the city, thereby ensuring that participants in the marathon would have an equitable experience. The inclusion of a legend in the figure provides further detail about which areas are associated with each train/bus station.

A marathon that is 42KM long was deemed to be in the range $41.5 \leq marathon\ length < 42.5$, where, for example, a total marathon of length 41.7KM would be acceptable. To facilitate the analysis, the graph was treated as an undirected graph, given that during marathon events, roads are often closed, allowing runners to move in both directions along one-way streets.

Once the seed nodes had been selected, the *voronoi* function found a list of all cycles which form a basis for cycles in each of the cells. A basis for cycles of a network is a minimal collection of cycles such that any cycle in the network can be written as a sum of cycles in the basis. Here, summation of cycles is defined as “exclusive or” of the edges [15]. For each pair of cycles, the program calculated the total length of the pair, including the connecting roads to and from each cycle and the seed node for that cell, forming a potential marathon for that cell. If this potential marathon was 42KM in length, it was used as that cell’s marathon. If the marathon was not 42KM, it was added to a list of potential marathons that did not meet the required length criteria. If no marathons of exactly 42KM were found at the end of this process, the function selected the closest potential marathon to 42KM (this only happened for the Guiseley Station cell, as discussed below).

The goal was to find 10 possible marathon paths for each of the 10 cells that were exactly 42KM long. This was not possible with my original seed node distribution, where only 9 out of 10 of the Voronoi cells had marathons of length 42KM. To resolve this issue, the program identified a third cycle of an appropriate length to add to the marathon for the Guiseley Station cell only. The function then printed the length of each marathon for each Voronoi cell, displayed in Table 3. The time taken to calculate 10 marathon routes for Leeds is around 278 seconds on average.

Using these marathons, the *display_voronoi* function displays the marathon routes on the map of Leeds. The function first calculated the nearest seed node for every node in the network and assigned a unique colour to each Voronoi cell. The function then assigned this colour to every edge based on the closest seed node, with the length of each road being used to calculate this. Each marathon route was displayed in red in Figure 7, with the seed nodes displayed in green. On the day of the marathons, each resident would arrive at their allotted train/bus station’s closest intersections and then complete the marathon by running through the marathon route, ending up back at the seed node.

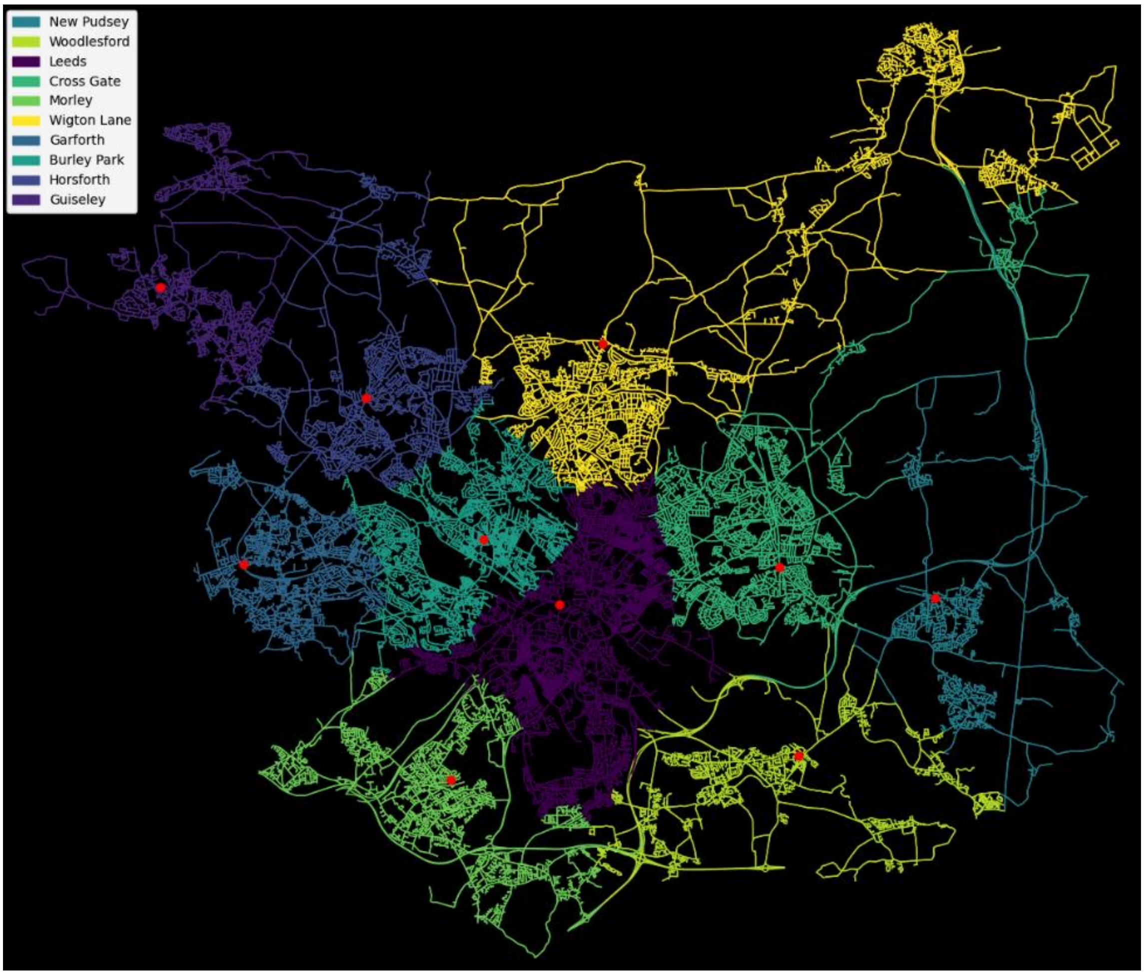

Figure 6: Map of Leeds with the selected seed nodes highlighted in red.

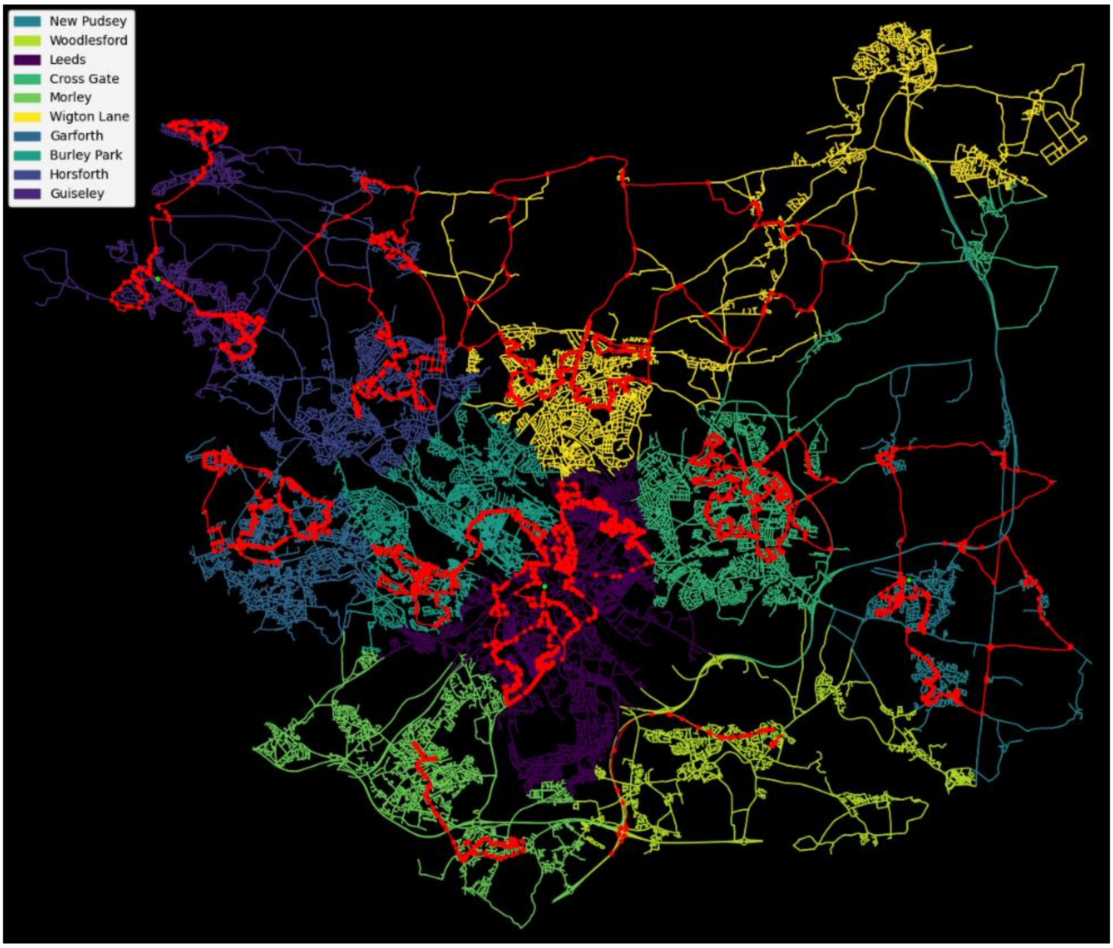

Figure 7: Map of Leeds with the selected marathons highlighted in red and seed nodes highlighted in green

I decided to incorporate features commonly seen in popular marathon tracks around the world. Some of the marathons, like the Guiseley Station marathon, use repeated roads to get to and from the cycles, while other marathons, like the Leeds Station marathon, are essentially just one long cycle. This feature is inspired by other international marathons, for example, the TCS London Marathon 2023, which also uses repeated roads to go to and from cycles [11], or the Los Angeles Marathon, where runners turn around and run back down part of the same course [12].

Some of the marathons, like the marathon displayed in the Morley Station cell (the bottom-left marathon in Figure 7, coloured green), also look shorter than the others because they incorporate two laps of essentially the same cycle. This was also inspired by other popular marathons around the world, such as the Torbay Half Marathon, which includes two laps [14], or the Barkley Marathons, which include as many as 5 laps [13].

Task D (TransE, PROV, PageRank)

To represent the provenance of important events in the road network of Leeds using the W3C PROV provenance data model standard, we need to understand the core concepts of this model; these are Entities, Activities, and Agents. An Agent is an entity that can influence an activity or be responsible for an activity. An Entity is a physical, digital, or conceptual object that is relevant to an activity, and an Activity is a thing that occurs over a period of time and acts on or with Entities.

In the context of road network events, Entities can be represented by physical objects such as the roads and intersections involved in accidents. Activities can be represented by events such as car accidents or marathons, which occur at a particular location on a specific date and time. Agents can be represented by people involved in accidents or marathons, including drivers, passengers, or runners. The diagram of this representation is displayed in Figure 7.

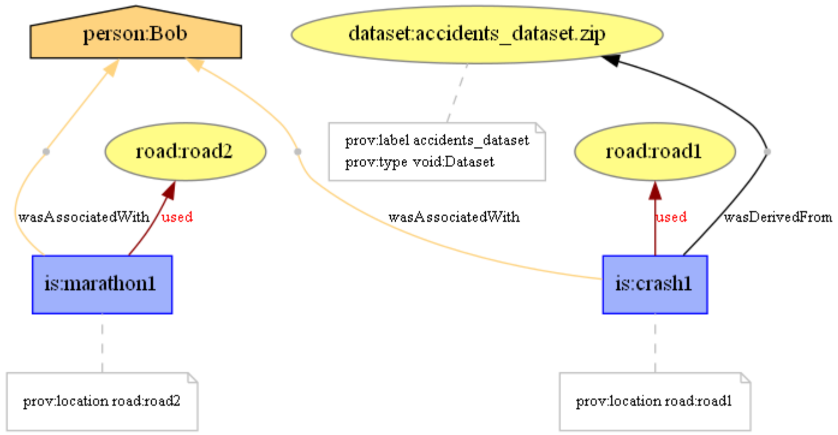

Figure 7: W3C PROV provenance diagram

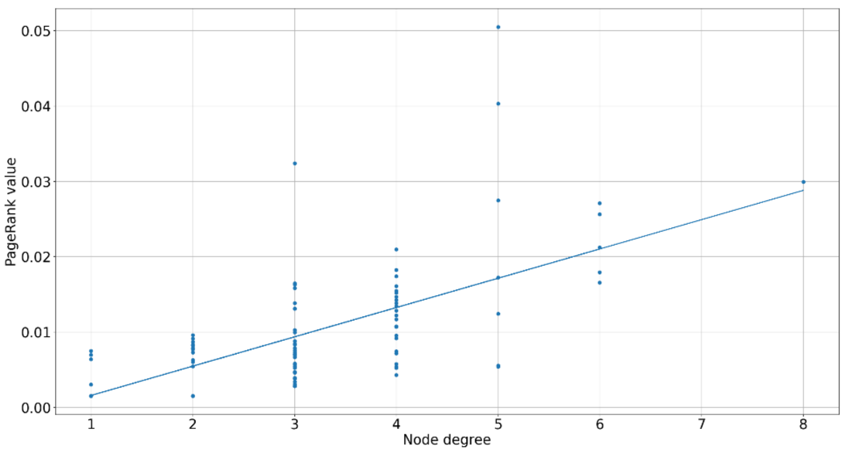

Figure 8: Node (intersection) degree against its PageRank value

The PageRank algorithm is a method for computing a ranking for every web page based on a network of the World Wide Web [5]. An adapted version of this algorithm is used as the basis for ranking Google search results, the most popular search engine in the world. The algorithm assigns a score to each web page based on the number of other web pages linking to said web page, also considering the quality of these web pages. A higher score indicates that that web page has a lot of high-quality links linking to it, meaning that that web page is considered important and relevant to the search query. The algorithm essentially calculates the probability that a user randomly clicking links will arrive at any given web page. These scores are calculated recursively, based on the scores of all of the linked pages, meaning that each page's score influences all of the pages that that page is linked to.

A page's quality is determined using multiple factors [6]. These include its relevance to the entered search query, meaning that if the page has a lot of words or phrases that are related to the search query, or the meaning derived from the search query; user engagement, meaning the longer users spend on web pages, and the higher the web pages click-through rates, the higher quality that web page is said to be; the accuracy, completeness, and trustworthiness of the web page's content; the web pages usability, meaning its loading speed and if it is mobile-friendly; and context, meaning the users past search history, search settings, and any similar relevant information. These factors constitute an extension of the original algorithm [5].

Google also inserts its advertisements into its search results, whereby if a campaign has bought adverts for a specific demographic, users of this demographic are likely to receive adverts that are related to their search queries [7].

Using my representation, if I computed the PageRank algorithm for each node in this network, meaning that each road/intersection would be assigned a score denoting how important it is, I would be able to rank the most important nodes in the network based on how many events (accidents or marathons) occur at each road. The higher a node's score, the more likely that events are to be clustered near that node. After computing the PageRank score for each intersection, I plotted this score against that intersection's degree, where a positive correlation is seen, shown in Figure 8.

TransE (Translating Embeddings) is a machine learning algorithm for knowledge graph completion and entity alignment [8], used in many applications such as recommender systems, natural language processing, and information retrieval. Entities and relationships are represented as embeddings, which are low-dimensional vectors. TransE, compared to other knowledge graph completion methods, is more efficient and scalable and can handle vast knowledge graphs with millions of entities and relationships.

TransE can be used with the Leeds car accident network to learn embeddings that capture the relationships (similarities) between accidents in the network which can be leveraged to identify similar accidents and cluster them together. This could help us identify patterns of accidents. This can help us gain insights into the patterns of accidents and identify the underlying factors that contribute to their occurrences.

Practically speaking, the learned embeddings can be used to address a range of problems related to road safety, such as identifying accident-prone areas, predicting accident hotspots, and recommending interventions to reduce the likelihood of accidents. If the embeddings reveal that a cluster of accidents is centred around a specific road, some examples of measures that can be taken by local councils to improve road safety include redesigning the road layout, installing traffic lights or speed cameras, or providing better pedestrian crossings. Similarly, if the embeddings suggest that certain types of accidents are more common than others, policymakers can tailor their interventions to target those specific types of accidents and mitigate their impact.

References

[1] "Leeds Population Density Map Viewer." n.d. https://www.arcgis.com/apps/mapviewer/index.html?webmap=d210539a84bf481caaf4720c841f6e28.

[2] "Every train station in Britain listed and mapped: find out how busy each one is," 2013. https://www.theguardian.com/news/datablog/2011/may/19/train-stations-listed-rail.

[3] "List of Busiest Railway Stations in West Yorkshire." 2022. Wikipedia. March 11, 2022. https://en.wikipedia.org/wiki/List_of_busiest_railway_stations_in_West_Yorkshire.

[4] J. d. B. Albert Meroño-Peñuela, *Network Data Analysis Topic 7: Spatial Networks in Context (slide 2),* King's College London.

[5] Page, Lawrence, Sergey Brin, Rajeev Motwani and Terry Winograd. "The PageRank Citation Ranking : Bringing Order to the Web." The Web Conference (1999).

[6] "Ranking Results – How Google Search Works." n.d. Google Search – Discover How Google Search Works. https://www.google.com/intl/en_uk/search/howsearchworks/how-search-works/ranking-results/.

[7] "About Display Expansion on Search Campaigns - Google Ads Help." n.d. Support.google.com. https://support.google.com/google-ads/answer/7193800?hl=en-GB.

[8] Bordes, Antoine, Xavier Glorot, Jason Weston, and Yoshua Bengio. 2013. "A Semantic Matching Energy Function for Learning with Multi-Relational Data." Machine Learning 94 (2): 233–59. https://doi.org/10.1007/s10994-013-5363-6.

[9] Veness, Chris. 2019. "Calculate Distance and Bearing between Two Latitude/Longitude Points Using Haversine Formula in JavaScript." Movable-Type.co.uk. 2019. https://www.movable-type.co.uk/scripts/latlong.html.

[10] Council, Leeds City. 2020. "Road Traffic Accidents." Www.data.gov.uk. July 22, 2020. https://www.data.gov.uk/dataset/6efe5505-941f-45bf-b576-4c1e09b579a1/road-traffic-accidents.

[11] "The Course." n.d. Virgin Money London Marathon. https://www.tcslondonmarathon.com/the-event/the-course.

[12] "LA Marathon Course." n.d. The McCourt Foundation. https://www.mccourtfoundation.org/pages/la-marathon-course.

[13] "Barkley Marathons." 2023. Wikipedia. April 3, 2023. https://en.wikipedia.org/wiki/Barkley_Marathons.

[14] "Torbay Half Marathon." n.d. Www.torbayhalfmarathon.co.uk. https://www.torbayhalfmarathon.co.uk/race_info.php.

[15] "Cycle_basis — NetworkX 3.1 Documentation." n.d. Networkx.org. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.cycles.cycle_basis.html.

Appendix

Source Code GitHub Link: https://github.com/Dan1elAnthony/k19012373_Leeds_Accidents/blob/main/k19012373.py

Source code starts from next page onwards…

k19012373.py

```python
# NDA Coursework 2
# Daniel Van Cuylenburg (k19012373)
# 11/04/2023
#
# Program that returns statistics about the Leeds road network, car accidents
# in Leeds (2009-2019), and plots 10 marathon routes through Leeds.
# Format follows the Google Python Style Guide.
#

# Imports
from networkx import Graph, get_node_attributes, diameter, check_planarity, voronoi_cells,
cycle_basis, path_weight, shortest_path_length, shortest_path, pagerank
from osmnx import graph_from_bbox, graph_from_place, graph_to_gdfs, basic_stats
from osmnx.plot import get_colors, plot_graph
from geopandas import GeoDataFrame, sjoin_nearest
from pandas import DataFrame, read_csv, concat
from numpy import polyfit, poly1d, arange
from spaghetti import Network, element_as_gdf
from seaborn import histplot
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
from shapely.geometry import Point, LineString

# Python Standard Library
from time import process_time

NORTH, SOUTH, WEST, EAST = 53.804, 53.794, -1.546, -1.536

class Main:
    """Class that downloads and processed a map of Leeds with car accident
    data.

    Attributes:
        leeds_graph (networkx MultiDiGraph): Graph representation of either the
            Leeds town centre or the whole of Leeds (drivable roads only).
        edges_gdf (geopandas GeoDataFrame): Edge (road) data of the road
            network of Leeds.
        nodes_gdf (geopandas GeoDataFrame): Node (intersection) data of the
            road network of Leeds.
        leeds_undirected_graph (networkx Graph): Undirected graph
            representation of the Leeds drivable road network.
        roads_gdf (geopandas GeoDataFrame): Road geometry data (coordinates of
            roads) of Leeds.
        accidents (geopandas GeoDataFrame): Leeds car accidents data
            (2009-2019).
        seeds (list of int (nodes)): Chosen seed nodes for the Voronoi diagram.
        marathons (list of lists of ints (nodes)) Chosen marathons for each
            Voronoi cell.
        places (list of str): Names of train/bus stations for each Voronoi
            cell.
        cells (dict): Voronoi cells. Keys are seed nodes, values are all the
            nodes that belong to that seed node's cell.
    """

    def __init__(self):
        """Inits Main class."""
```

```
        # Leeds Town Centre 1 square KM analysis.
        self.generate_city_graph("1sqKM")
        self.generate_roads()
        self.print_characteristics()
        self.import_accidents()
        self.print_map()
        self.count_accidents()
        self.count_adjacent_accidents()
        # self.plot_accidents()
        self.investigate_intersections()
        # self.plot_intersection_fractions()
        # self.plot_pagerank()

        # Whole of Leeds marathon analysis.
        self.generate_city_graph("all")
        self.generate_roads()
        start = process_time()
        self.select_seeds()
        self.voronoi()
        print("Time taken to calculate 10 marathon routes:",
              process_time() - start, "seconds")
        self.display_voronoi()

    def generate_city_graph(self, graph_type):
        """Downloads the OpenStreetMap data for the specified "graph_type".

        Args:
            graph_type (str): 1 square KM Leeds Town Centre or the whole of
                Leeds.
        """
        if graph_type == "all":
            city = "Leeds, UK"
            self.leeds_graph = graph_from_place(city, network_type="drive")
        else:
            self.leeds_graph = graph_from_bbox(
                north=NORTH, south=SOUTH, west=WEST, east=EAST,
                network_type="drive")
        self.nodes_gdf, self.edges_gdf = graph_to_gdfs(self.leeds_graph)
        self.leeds_undirected_graph = Graph(self.leeds_graph)

    def generate_roads(self):
        """Adds geometry data for any missing roads.
            Note: Taken and adapted from Week 8 Python Notebook.
        """
        x_values = get_node_attributes(self.leeds_graph, "x")
        y_values = get_node_attributes(self.leeds_graph, "y")
        graph_with_geometries = list(self.leeds_graph.edges(data=True))
        # Iterates through the edges and, where missing, adds a geometry
        # attribute with the line between start and end nodes.
        for e in graph_with_geometries:
            if not "geometry" in e[2]:
                e[2]["geometry"] = LineString([
                    Point(x_values[e[0]], y_values[e[0]]),
                    Point(x_values[e[1]], y_values[e[1]])])
        # Declares a GeoDataFrame with each road's geometry data.
        road_lines = [x[2] for x in graph_with_geometries]
        self.roads_gdf = GeoDataFrame(DataFrame(road_lines))
        # Sets GeoDataFrame to latitude/longitude coords system.
        self.roads_gdf.set_crs("EPSG:4326", inplace=True)
        # Assigns a given road's two intersection nodes to itself.
```

```
        self.roads_gdf["nodes"] = ""
        def get_nodes(row):
            nodes = self.edges_gdf.iloc[row.name].name
            return (nodes[0], nodes[1])
        self.roads_gdf["nodes"] = self.roads_gdf.apply(get_nodes, axis=1)

    def print_characteristics(self):
        """Calculates and prints statistics about the selected road network."""
        area = ((NORTH - SOUTH) * 111.1) * ((EAST - WEST) * 111.1)
        print("Total area:", area , "square KM")
        statistics = basic_stats(self.leeds_graph, area=area*1000000)
        print("Number of intersections:", statistics["intersection_count"])
        print("Number of roads:",
              sum(statistics["streets_per_node_counts"].values()))
        print("Average number of roads per intersection:", statistics["k_avg"])
        print("Average number of streets per node:",
              statistics["streets_per_node_avg"])
        print("Total length of streets:",
              statistics["street_length_total"], "metres")
        print("Average length of a street:",
              statistics["street_length_avg"], "metres")
        print("Average number of intersections per square KM:",
              statistics["intersection_density_km"])
        print("Average number of streets per square KM:",
              statistics["street_density_km"])
        print("Spatial diameter:",
              diameter(self.leeds_undirected_graph, weight="length"))
        print("Sum of direct distances between nodes:",
              statistics["edge_length_total"], "metres")
        print("Average circuitry of network:", statistics["circuity_avg"])
        print("Is the network planar?", check_planarity(self.leeds_graph)[0])

    def import_accidents(self):
        """Imports Leeds car accidents 2009-2019, only keeping accidents in the
            selected area.
        """
        url_list = ["""8e6585f6-e627-4258-b16f-ca3858c0cc67/Traffic%2520accidents_2019_Leeds.csv""",
                    """8c100249-09c5-4aac-91c1-9c7c3656892b/RTC%25202018_Leeds.csv""",
                    """ca7e4598-2677-48f8-be11-13fd57b91640/Leeds_RTC_2017.csv""",
                    """b2c7ebba-312a-4b3d-a324-6a5eda85fa5b/Copy%2520of%2520Leeds_RTC_2016.csv""",
                    """df98a6dd-704e-46a9-9d6d-39d608987cdf/2015.csv""",
                    """fa7bb4b9-e4e5-41fd-a1c8-49103b35a60f/2014.csv""",
                    """56550461-ea6c-47d7-be61-73339b132547/2013.csv""",
                    """6ff5a09b-666a-4420-92ea-b6817b4a0f5c/2012.csv""",
                    """9204d06c-8e43-42d3-9ffa-87d806661801/2011.csv""",
                    """1ead4f5f-3636-4b8f-830c-7d2cc6f16084/2010.csv""",
                    """288d2de3-0227-4ff0-b537-2546b712cf00/2009.csv"""]
        # For each URL (csv file), download that years car accident data.
        # Standardizes the grid reference column names across all the files.
        accidents_df = DataFrame()
        for index, url in enumerate(url_list):
            csv = read_csv("""https://datamillnorth.org/download/road-traffic-accidents/""" + url,
                           encoding="unicode_escape", low_memory=False).rename(
                               columns={"Grid Ref: Easting": "Easting",
                                        "Grid Ref: Northing": "Northing"})
            csv["Year"] = range(2019, 2008, -1)[index]
            accidents_df = concat([accidents_df, csv])
        # Ensures each row represents a unique accident.
```

```
        accidents_df = accidents_df.drop_duplicates(subset="Reference Number")
        accidents_df.reset_index(inplace=True)
        # Turns accidents into a GeoDataFrame with a geometry column.
        accident_points = GeoDataFrame(
            geometry=[Point(xy) for xy in zip(accidents_df["Easting"],
                                              accidents_df["Northing"])],
            crs="EPSG:27700")
        accident_points["Year"] = accidents_df["Year"]
        # Converts easting/northing into latitude/longitude coords systems.
        accident_points.to_crs("EPSG:4326", inplace=True)
        # Filters for only accidents in the denoted ~1sqKM area.
        self.accidents = accident_points[accident_points.geometry.within(
            self.nodes_gdf.unary_union.convex_hull)]
        print("Number of unique accidents in the area:", len(self.accidents))

    def print_map(self):
        """Displays a map of the selected area (Leeds Town Centre) with all of
            the car accidents plotted on the map where they happened.
        """
        # Creates a graph of roads only.
        roads_network = Network(in_data=self.roads_gdf)
        nodes_df, edges_df = element_as_gdf(roads_network, vertices=True,
                                            arcs=True)
        # Snaps the accidents onto the roads graph.
        roads_network.snapobservations(self.accidents, "accidents")
        # Plots the roads.
        base_network = edges_df.plot(color="k", zorder=0, figsize=(10, 10))
        # Creates a GeoDataFrame from the accidents.
        roads_accidents_gdf = element_as_gdf(
            roads_network, pp_name="accidents", snapped=True)
        accidents = self.accidents.reset_index()
        roads_accidents_gdf["Year"] = accidents["Year"]
        # Normalizes the accident year data to lie between 0 and 1.
        plt.Normalize(roads_accidents_gdf["Year"].min(),
                      roads_accidents_gdf["Year"].max())
        # Plots and displays the snapped accident locations with colors based
        # on the year of occurrence.
        roads_accidents_gdf.plot(
            column="Year",
            cmap=LinearSegmentedColormap.from_list(
                "custom_map", ["#ADD8E6", "#00008B"], 10),
            legend=True,
            classification_kwds=dict(
                bins=list(roads_accidents_gdf["Year"].unique())),
            markersize=200,
            marker="x",
            alpha=0.8,
            zorder=1,
            ax=base_network
        )
        print("\nClose the map to continue.\n")
        plt.show()

    def count_accidents(self):
        """Counter the number of accidents per road and per intersection."""
        self.roads_gdf["accidents"] = 0
        self.nodes_gdf["accidents"] = 0
        self.accidents[["node"]] = ""
        # Performs a spatial join between accidents and roads.
        roads_to_join = self.roads_gdf[["geometry"]].copy()
```

```
        joined = sjoin_nearest(self.accidents.to_crs(crs="EPSG:27700"),
                               roads_to_join.to_crs(crs="EPSG:27700"),
                               how="left")
        # Sums the number of accidents per road.
        sum = joined.groupby("index_right").size()
        # Assigns the counts to the roads GeoDataFrame.
        self.roads_gdf.loc[sum.index, "accidents"] = sum.values

        # Performs a spatial join between accidents and intersections.
        nodes_to_join = self.nodes_gdf[["geometry"]].copy()
        joined = sjoin_nearest(self.accidents.to_crs(crs="EPSG:27700"),
                               nodes_to_join.to_crs(crs="EPSG:27700"),
                               how="left")
        # Sums the number of accidents per intersection.
        sums = joined.groupby("index_right").size()
        # Assigns the counts to the nodes GeoDataFrame.
        self.nodes_gdf.loc[sums.index, "accidents"] = sums.values
        # Assigns the id of the node to each accident.
        self.accidents["node"] = joined["index_right"].values

    def count_adjacent_accidents(self):
        """Counts the number of adjacent accidents per road."""
        self.roads_gdf["adj_accidents"] = 0
        for index, road1 in self.roads_gdf.iterrows():  # For each road.
            adj_accidents = 0
            remaining = self.roads_gdf.drop(index)
            # For each adjacent road if the two roads are connected, adds that
            # adjacent roads accidents.
            for _, road2 in remaining.iterrows():
                if road1["nodes"][0] == road2["nodes"][0]:
                    adj_accidents += road2["accidents"]
                if road1["nodes"][0] == road2["nodes"][1]:
                    adj_accidents += road2["accidents"]
                if road1["nodes"][1] == road2["nodes"][0]:
                    adj_accidents += road2["accidents"]
                if road1["nodes"][1] == road2["nodes"][1]:
                    adj_accidents += road2["accidents"]
            self.roads_gdf.at[index, "adj_accidents"] = adj_accidents

    def plot_accidents(self):
        """Plots a scatter plot the the number of accidents per road against
            the number of adjacent accidents for that road.
        """
        x = self.roads_gdf["accidents"]
        y = self.roads_gdf["adj_accidents"]
        plt.scatter(x, y, zorder=2)
        plt.xticks(range(0, 30, 1), fontsize=15)
        plt.yticks(range(0, 130, 5), fontsize=15)
        plt.ylabel("Number of Accidents on Adjacent Roads", fontsize=25)
        plt.xlabel("Number of Accidents on a Given Road", fontsize=25)
        plt.plot(x, poly1d(polyfit(x, y, 1))(x))
        plt.grid(zorder=1)
        plt.show()

    def investigate_intersections(self):
        """Calculates the road fraction for each accident, where
            1 = midpoint of road, 0 = intersection of road.
        """
        # Perform a spatial join to find the nearest road for each accident.
        roads_for_join = self.roads_gdf[["geometry", "length"]].copy()
```

```python
        joined = sjoin_nearest(
            self.accidents.to_crs(crs="EPSG:27700"),
            roads_for_join.to_crs(crs="EPSG:27700"),
            distance_col="distance", how="left")
        # Drops any duplicated accidents.
        joined = joined.drop_duplicates(subset = "geometry")
        # Calculate the road fraction for each accident.
        joined["road_fraction"] = (joined["distance"]) / (joined["length"] / 2)
        # Assign the road fraction back to the "self.accidents" dataframe.
        self.accidents["road_fraction"] = joined["road_fraction"]

    def plot_intersection_fractions(self):
        """Plots a boxplot and histogram of the distances of the accidents from
            the intersections (road fractions calculated in the
            "investigate_intersections" function).
        """
        # "sjoin_nearest" spatial join does not work properly for all of the
        # accidents, so remove the accidents that have been calculated
        # incorrectly (a small minority).
        accidents = self.accidents.drop(
            self.accidents[self.accidents.road_fraction > 1].index).dropna(
                subset="road_fraction")
        # Plots a box plot of the road fractions.
        plt.boxplot(accidents["road_fraction"])
        plt.xticks([])
        plt.yticks(arange(0, 1.1, 0.1), fontsize=25, rotation=90)
        plt.ylabel("Road Fraction\n(1 = midpoint of road, 0 = intersection of road)",
                   fontsize=25)
        plt.show()
        # Plots a histogram of the road fractions.
        histplot(data=accidents["road_fraction"], color="r", alpha=0.5,
                     element="bars", kde=True, binwidth=0.01)
        plt.yticks(range(0, 65, 10), fontsize=25)
        plt.xticks(arange(0, 1.1, 0.1), fontsize=25)
        plt.ylabel("Number of Accidents", fontsize=25)
        plt.xlabel("Road Fraction (1 = midpoint of road, 0 = intersection of road)",
                   fontsize=25)
        plt.show()

    def plot_pagerank(self):
        """Calculates and plots pagerank of nodes.
            Note: Taken and adapted from 7CUSMNDA week 10 exercise solutions.
        """
        pagerank_dict = pagerank(self.leeds_graph, alpha=0.9)
        pagerank_sorted_desc = dict(sorted(pagerank_dict.items(),
                                           key=lambda item: item[1],
                                           reverse=True))
        node_degree = {k: v for k, v in self.leeds_graph.degree(
            pagerank_sorted_desc.keys())}

        x = list(node_degree.values())
        y = list(pagerank_sorted_desc.values())
        fig, ax = plt.subplots()
        ax.scatter(x, y, zorder=2)
        plt.yticks(arange(0, 0.07, 0.01), fontsize=25)
        plt.xticks(range(0, 10, 1), fontsize=25)
        plt.ylabel("PageRank value", fontsize=25)
        plt.xlabel("Node degree", fontsize=25)
        plt.plot(x, poly1d(polyfit(x, y, 1))(x))
        plt.grid(zorder=1)
```

```
        plt.show()

    def select_seeds(self):
        """Selects 10 seed nodes (intersections) that have been preselected as
            latitude/longitude coordinates of the most popular train/bus
            stations.
        """
        self.seeds = []
        self.marathons = []
        # Station names for each cell.
        self.places = ["Leeds", "Guiseley", "Horsforth", "New Pudsey",
                       "Garforth", "Burley Park", "Cross Gates", "Morley",
                       "Woodlesford", "Wigton Lane"]
        # Train station coordinates.
        train_stations = [Point(-1.5474, 53.7950), Point(-1.71767, 53.87547),
                          Point(-1.63, 53.8476), Point(-1.68207, 53.80527),
                          Point(-1.38464, 53.79672), Point(-1.57906, 53.81157),
                          Point(-1.4516, 53.8047), Point(-1.5931, 53.75065),
                          Point(-1.4437, 53.7570), Point(-1.5279, 53.8612)]
        # Turns the coordinates into a GeoDataFrame.
        train_stations_gdf = GeoDataFrame({"geometry": train_stations},
                                          crs="EPSG:4326")
        # Performs a spatial join between station coordinates and
        # intersections.
        joined = sjoin_nearest(train_stations_gdf.to_crs(crs="EPSG:27700"),
                               self.nodes_gdf.to_crs(crs="EPSG:27700"),
                               how="left")
        # For each of the closest nodes, adds this node to "self.seeds".
        for node in joined["index_right"]:
            self.seeds.append(self.nodes_gdf.loc[node].name)
        # Voronoi cells centered at "self.seeds" using the lengths of roads as
        # the shortest-path distance metric. Keys are each node in the network,
        # values are the seed node that is closest to it.
        self.cells = voronoi_cells(self.leeds_undirected_graph,
                                   self.seeds, weight="length")

    def voronoi(self):
        """Calculates 10 Voronoi cells based on "self.seeds". Calculates 10
            42KM marathons (trails) for each cell, saved in "self.marathons".
        """
        # For each seed node.
        for seed_index, seed_node in enumerate(self.seeds):
            # Converts from MultiGraph to Graph.
            subnetwork = Graph(
                self.leeds_graph.subgraph(self.cells[seed_node]))
            all_nodes = list(subnetwork.nodes)
            # Returns a list of cycles which form a basis for cycles of the
            # subnetwork.
            all_cycles = cycle_basis(subnetwork)
            # For each cycle in the subnetwork, finds length of that cycle.
            cycle_lengths = []
            for cycle in all_cycles: cycle_lengths.append(
                path_weight(subnetwork, cycle, weight="length"))
            # For each node in the subnetwork, find the shortest path from the
            # current seed node to that node.
            shortest_paths = []
            for node in all_nodes: shortest_paths.append(
                shortest_path(subnetwork, source=seed_node,
                              target=node, weight="length"))
            # Constraints for each cell to get the best marathon (attempts to
```

```python
            # avoid repeated cycles for a more 'scenic' marathon trail). Can be
            # used to configure the algorithm and get different marathon trails
            # for each station's cell.
            constraints = [40000, 1, 40000, 37000, 39000,
                           35000, 37500, 5000, 1, 40000]
            # Loops over cycles twice, finding the best pair of cycles to use
            # as marathon trails (meaning 42KM length, if possible).
            potential_marathons = []
            potential_marathon_lengths = []
            found = False
            # For each cycle.
            for i, cycle_length_1 in enumerate(cycle_lengths):
                cycle1_nodes = all_cycles[i]
                # For each remaining cycle.
                for j, cycle_length_2 in enumerate(cycle_lengths[i+1:]):
                    cycle2_nodes = all_cycles[j]
                    full_length = cycle_length_1 + cycle_length_2
                    full_marathon = cycle1_nodes + cycle2_nodes
                    # If both cycle's lengths are within the constraints.
                    if constraints[seed_index] < full_length < 42500:
                        # For each cycle's list of nodes.
                        for nodes in [cycle1_nodes, cycle2_nodes]:
                            shortest_distance = float("inf")
                            # Finds the closest node in the cycle to the seed
                            # node, adds this path and 2 * its length to the
                            # marathon trail.
                            for node in nodes:
                                path = shortest_paths[all_nodes.index(node)]
                                from_seed_length = path_weight(
                                    subnetwork, path, weight="length")
                                if from_seed_length < shortest_distance:
                                    shortest_distance = from_seed_length
                                    selected_path = path
                            full_marathon.extend(selected_path)
                            full_length += 2 * shortest_distance
                        # Adds each found marathon to a list in case we don't
                        # find one within the constraints and need to pick the
                        # best one later (only happens for Guiseley anyway;
                        # read report for more information).
                        potential_marathons.append(full_marathon)
                        potential_marathon_lengths.append(full_length)
                    # If we have found a marathon within the constraints, then
                    # break the for loops and just use that one. This makes the
                    # algorithm more computationally efficient while still
                    # staying within the constraints.
                    if 41500 <= full_length < 42500:  # If marathon is 42KM.
                        potential_marathons = [full_marathon]
                        potential_marathon_lengths = [full_length]
                        found = True
                        break
                if found: break
            # Calculates the best marathon found from the loops above by
            # finding the closest marathon length to 42KM.
            min_length = min(potential_marathon_lengths,
                             key=lambda x:abs(x-42000))
            marathon_index = potential_marathon_lengths.index(min_length)
            marathon = potential_marathons[marathon_index]
            marathon_length = min_length
            # The Guiseley (train station) cell is the only cell with a
            # marathon of length less than 42KM. Therefore, for this marathon
```

```
            # only, add a third cycle to make the total length 42KM. Similar to
            # the above algorithm, but only looping once over cycles.
            if seed_index in [1]:  # If the current seed node is Guiseley.
                potential_marathons = []
                potential_marathon_lengths = []
                length_needed = 42000 - min_length
                # For each cycle.
                for i, cycle_length in enumerate(cycle_lengths):
                    # If the cycle can be added without going over 42KM.
                    if cycle_length < length_needed:
                        cycle_nodes = all_cycles[i]
                        shortest_distance = float("inf")
                        # Finds the closest node in the cycle to the seed node,
                        # adds this path and 2 * its length to the marathon
                        # trail.
                        for node in cycle_nodes:
                            path = shortest_paths[all_nodes.index(node)]
                            from_seed_length = path_weight(subnetwork, path,
                                                           weight="length")
                            if from_seed_length < shortest_distance:
                                shortest_distance = from_seed_length
                                selected_path = path
                        potential_marathons.append(cycle_nodes + selected_path)
                        potential_marathon_lengths.append(
                            cycle_length + 2 * shortest_distance)
                # Finds the best cycle out of all the cycles to add to the
                # marathon (based on the closest combination of cycles to
                # a total length of 42KM).
                min_length2 = min(potential_marathon_lengths,
                                  key=lambda x:abs(x-length_needed))
                marathon_index = potential_marathon_lengths.index(min_length2)
                marathon.extend(potential_marathons[marathon_index])
                marathon_length += min_length2
            # Adds this seed node's marathon to the final "self.marathons"
            # list.
            self.marathons.extend(marathon)
            print(self.places[seed_index], "marathon length:", marathon_length)

    def display_voronoi(self):
        """Displays a map of the calculate voronoi cells and marathons, with
            the edges in each cell being different colors, the seed nodes
            green, the marathons red.
            Note: Taken and adapted from 7CUSMNDA week 6 exercise solutions.
        """
        color_order_places = ["New Pudsey", "Woodlesford", "Leeds",
                              "Cross Gate", "Morley", "Wigton Lane",
                              "Garforth", "Burley Park", "Horsforth",
                              "Guiseley"]
        # Keys are seed nodes, values are a list of nodes that are closest to
        # that seed node.
        node_seed_dict = {v: key for key,
                          value in self.cells.items() for v in value}
        # Keys are seed nodes, values are that seed nodes mapped color.
        seed_colors = dict(zip(self.seeds, get_colors(len(self.seeds))))
        # Unreachable nodes/edges to be invisible.
        seed_colors["unreachable"] = (0, 0, 0, 1)
        # Keys are nodes, values are their colors.
        node_color_dict = {node: seed_colors[
            node_seed_dict[node]] for node in self.leeds_graph.nodes}
        # List of colors corresponding to the networks edges.
```

```
        edge_colors = self.map_edge_color_from_node(node_seed_dict,
                                                    node_color_dict)
        # Retrieves a list of all unique colors being used for each cell.
        unique_colors = []
        for c in edge_colors:
            if c not in unique_colors: unique_colors.append(c)
        # Turns 42KM marathon edges red.
        for index, edge in enumerate(list(self.leeds_graph.edges)):
            if edge[0] in self.marathons and edge[1] in self.marathons:
                edge_colors[index] = (1, 0, 0, 1)
        # Assigns node colors.
        node_colors = []
        for node in list(self.leeds_graph.nodes):
            # If seed node, green.
            if node in self.seeds: node_colors.append((0, 1, 0, 1))
            # If marathon path node, red.
            elif node in self.marathons: node_colors.append((1, 0, 0, 1))
            # If any other node, invisible.
            else: node_colors.append((0, 0, 0, 0))
        # Keys are colors, values are names of that cell's train station.
        color_labels = dict(zip(unique_colors, color_order_places))
        # Plots the Voronoi cells with the assigned colors.
        figure, axis = plot_graph(self.leeds_graph, edge_color=edge_colors,
                                  node_color=node_colors, bgcolor="k",
                                   show=False)
        # Adds a legend based on the colors associated with the stations.
        axis.add_artist(axis.legend(handles=[plt.Rectangle((0,0),1,1, color=color) for color in color_labels.keys()],
                                    labels=color_labels.values()))
        # Displays the plot.
        plt.show()

    def map_edge_color_from_node(self, node_seed_dict, node_color_dict):
        """Assigns a color to each edge based on its closest seed node.
            Note: Taken and adapted from 7CUSMNDA week 6 exercise solutions.

        Args:
            node_seed_dict (dict): Keys are seed nodes, values are a list of
                nodes that are closest to that seed node.
            node_color_dict (dict): Keys are nodes, values are their colors.

        Returns:
            List of strings: Assigned color of each edge.
        """
        edge_colors = []
        for e in list(self.leeds_graph.edges):
            color_pair = [node_color_dict[e[0]], node_color_dict[e[1]]]
            # If node is unreachable, make its edges invisible.
            if (0, 0, 0, 1) in color_pair:
                color_pair.remove((0, 0, 0, 1))
                edge_colors.append(color_pair[0])
            # Else if both nodes are the same color, the edge between them
            # should be that color.
            elif color_pair[0] == color_pair[1]:
                edge_colors.append(color_pair[0])
            # Else, based on which node is closer to the seed node, assign that
            # nodes color to the edge.
            else:
                len_0 = shortest_path_length(self.leeds_undirected_graph,
                                             node_seed_dict[e[0]], e[0],
```

```
                                            weight="length")
                len_1 = shortest_path_length(self.leeds_undirected_graph,
                                            node_seed_dict[e[1]], e[1],
                                            weight="length")
                if len_0 <= len_1:
                    edge_colors.append(color_pair[0])
                else:
                    edge_colors.append(color_pair[1])
        return edge_colors


Main()

```