Final Report: Navigating Portland

Authors: Emma Morse, Shuiming Chen, Amanda Haskell

Introduction

Question:

Given Portland roads as the network we travel on. Can we determine optimal locations for a given number of bus stops, such that we are minimizing distance to amenities (schools, healthcare and grocery shops)?

Shuiming chen:

Dijkstra's algorithm is used to find the shortest path between nodes in a graph, so maybe it is a great idea to apply this algorithm in our daily life.

Suppose we named the starting point as the student's living area, and Hannaford Supermarket, Walmart, Trader Joe's, Whole Foods Market, Roux institute, Maine Medical Center, Portland Museum of Art, Walgreens, CVS, Workout Anytime, Portland International Jetport, Portland Head Light, Old Port etc. as the end points, so that we can make a graph that covers most of public amenities that a roux institute student might going. And the weight edges of the map will be using the road lengths.

Why bus routes instead of driving routes? We are trying to apply Dijkstra's algorithm in our real daily life. On the one hand, some students may not own a car, on the other hand, if students can drive a car, it would be kind of/relatively meaningless to apply this algorithm in Portland city because they can drive anywhere straightforward. When using bus routes, there are some amenities that the students cannot reach out straight, it will help to build the map that the start point may need some intersection point to reach out other end points.

After we finished building the graph, and applying dijkstra's algorithm in our project, there are some facts that we may dismiss before, such as this algorithm cannot totally solve all the problems we met in this project. Then we tried to use floyd warshall algorithm, but it still cannot fully solve them all. Until we try to apply TSP in our project, it did help to figure out what we want to do in the beginning of our thoughts in this project. So our final workout is slightly different from what we have proposed before, but the main purpose was the same.

Amanda:

Public transportation is a valuable resource to students, as discussed above. It's also a valuable resource to those who have limited access to other modes of transportation. These may be physical limitations, financial limitations, or any number of other challenges. All people have basic needs that must be met. This is a fundamental principle I frequently utilize as a healthcare worker, based on Maslow's hierarchy of needs[1]. First physiological needs must be met (food, shelter), then safety needs (health, family) and so on. We have chosen to develop an optimized public transportation route including locations like grocery stores, healthcare facilities, etc... as a way to meet the needs of the residents of Portland.

Emma:

Greater Portland's primary public transportation system: GPMetro is an organization that is very tightly funded. In 2022 GPMetro's annual budget was $13,144,976[2] and needed to seek out federal and state funding to maintain, upgrade or expand facilities. Because of this GPMetro struggles to expand or add routes due to the added cost of employees, buses and facility space. Nevertheless, the bus system has an annual ridership of almost 1.25 million (still recovering from COVID-19) making it a critical transportation option for Southern Mainers. Having researched this, it is clear that this is a precious resource that does not get the funding it needs. Being a pedestrian first resident of Portland, I have on many occasions attempted to use the bus system and found it to be both infrequent and bus-stops inconveniently placed for my desired destination. Without learning to drive and accepting the cost of car ownership there are unfortunately still areas of Portland and surrounding towns that are inaccessible to me. We cannot necessarily explore the issue of frequency in this project, however; I am interested to learn if the routes have been designed optimally. Are there alternative routes that are more efficient and hence cheaper to maintain?

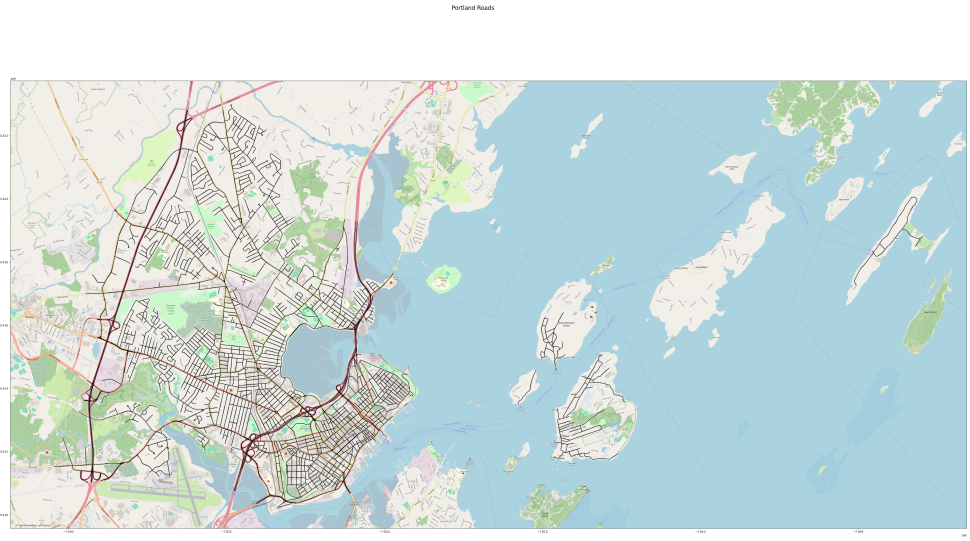
Problem Formulation, Implementation and Experimentation

Data Collection

We began our analysis by collecting data on the road systems in Maine. Data was retrieved from the Maine DOT open data arcgis server, and filtered to include only roads in the city of Portland. [3] The data was saved such that location geometries were transformed into a X,Y format on a 2D plane- this was accomplished by using the EPSG 3857 coordinate system.

Link to relevant source file

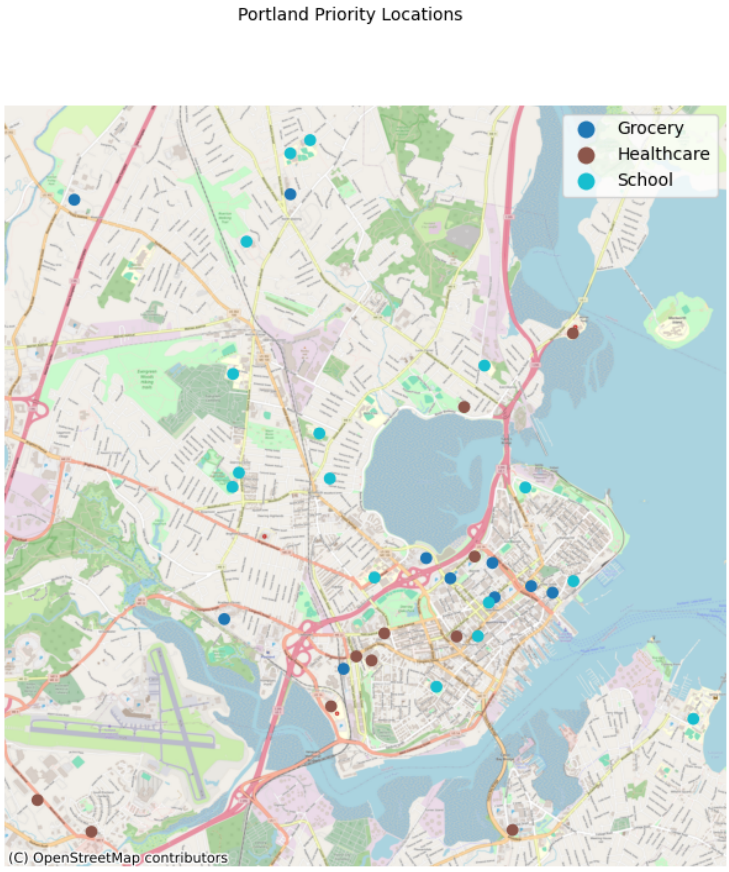
Portland Road Data Mapped



Next we gathered data on priority locations utilizing Google maps[4]. Location data for grocery stores, schools, and healthcare facilities were retrieved in WKT format, downloaded as separate csv files, then merged into one geodataframe. This location data was projected onto the same coordinate system as the road MaineDOT data.

Link to relevant source file

Priority Locations Data Mapped



Data Transformation

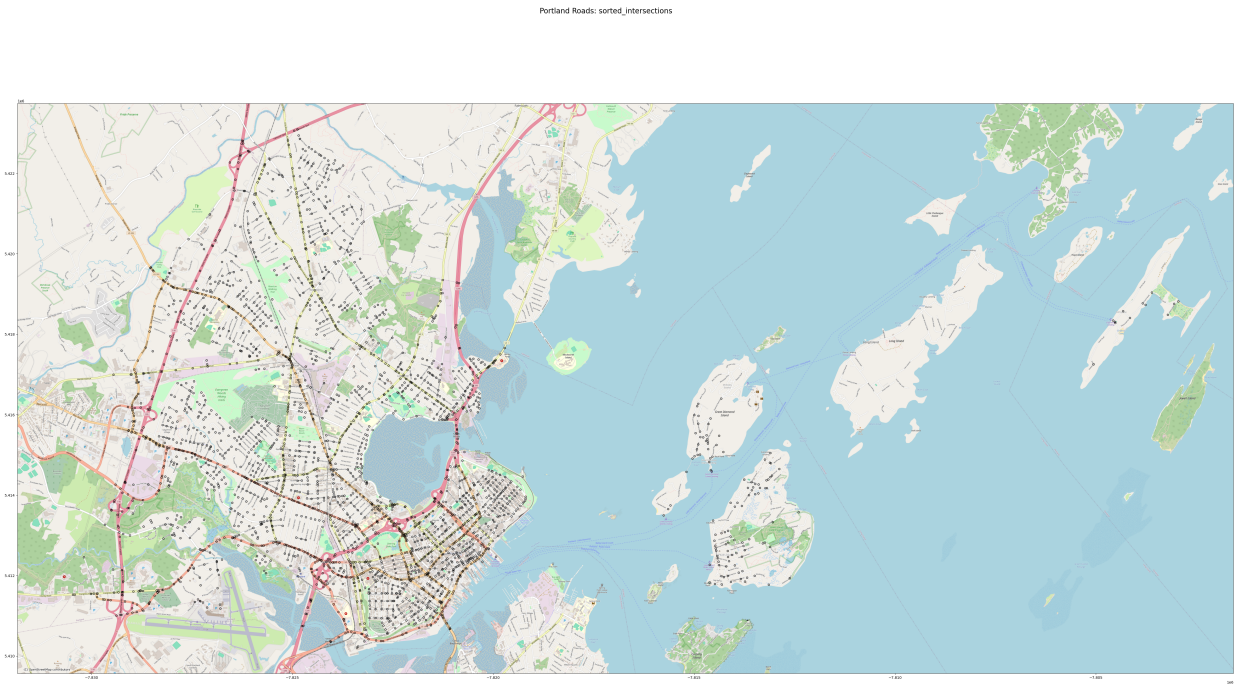
The data collected from the MaineDOT contained road locations as linestrings. Our goal was to transform that data into a graph format so we could utilize the graph traversal algorithms learned in Module 5.

Nodes for the graph were created from road intersections. This was accomplished by extracting the beginning and ending coordinates of each road, creating a node at each of those locations if such as node did not already exist, or updating nodes that did already exist. Weighted edges were created from the road segments between intersections, with a weight equal to the length of said segment.

Nodes and edges were stored in an adjacency list format, utilizing a nested Python dictionary data structure. The node(intersection) point coordinates were used as keys, with a dictionary as a value. The value dictionary includes the "neighbors" key, which is associated with a list of adjacent nodes(intersections). Additionally, road indexes, number or roads, and geometry data is stored in the nested dictionaries.

Link to relevant source file

Intersection Data Mapped

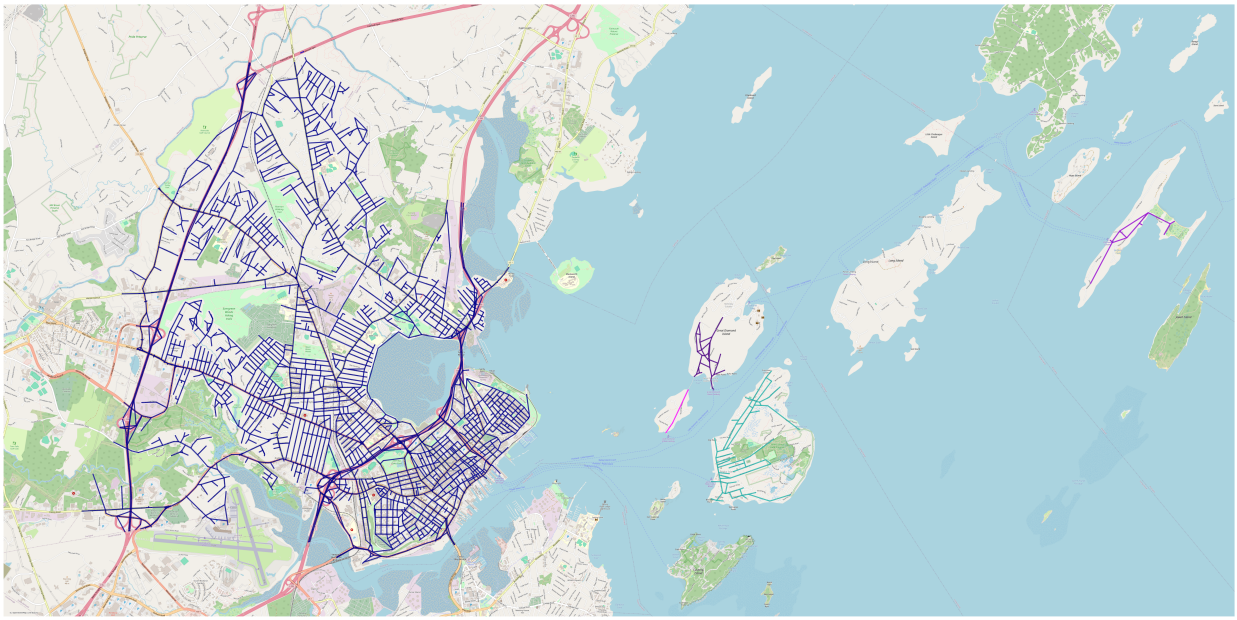


Next, a graph was created utilizing the Networkx Python package. Networkx is a package designed specifically for building, utilizing and manipulating complex graphs/network structures.[5]

This gave us an undirected, weighted graph with parameters shown below. As explained above, nodes are road intersections, edges are road segments between intersections, and edge weights are the road segment lengths.

Link to relevant source file

Road Data Graph Mapped



Graph Information

```
Number of roads:  3607
Number of intersections:  2735

Graph information:
Number of nodes:  2735
Number of edges:  3548
Number of connected components:  7
max degree:  8
min degree:  1
```

Next we needed to identify which graph nodes to include in our ideal public transportation route.

We judged that the graph produced from the roads was dense enough that all priority locations would be within reasonable walking distance from a road intersection, so the planimetric distance from our priority location coordinates to the intersection coordinates would be a reasonable heuristic to use find priority graph nodes.

We utilized some of the principles learned in Module 8- dynamic programing and Modules 6 and 7- greedy algorithms to create an algorithm that compared each single high-priority point to all our graph nodes (storing each value rather than iterating), and ultimately selecting the nodes with the minimum distance from each priority location as the representative node for that location. This gave us a list of 32 high priority nodes to include in our path.

Link to relevant source file

Shortest Path Analysis

Our original plan to find the optimal transportation route connecting priority locations was to use a version of a shortest path algorithm selecting minimum distances between nodes.

We began by applying Dijkstra's algorithm via the `Networkx.shorteset_path` method, selecting a source node from our list of priority nodes. We quickly realized, while this will give us the shortest path between each other priority node and the source node, it did not generate a path that included ALL of the priority nodes.

We considered applying Dijkstra's in a greedy stepwise manner: select the shortest distance between nodes in the priority node list, then select the node with the shortest path from the second node and so on, iterating through the remaining nodes in the priority node list. We realized, however, that this was not guaranteed to give us the shortest path between all nodes of the graph.

We then considered finding all the simple paths in the graph, filtering by only paths including our priority nodes, and selecting the path with the minimum cost.

While this would have likely given us the optimal answer, we utilized the topics learned from Module 1 and judged the time complexity to be prohibitive as our graph has 2735 nodes and 3548 edges. This method would have a time complexity of O(n!) where n = number of nodes in the graph.

Next we examined utilizing the Floyd-Warshall algorithm to generate the shortest path between all pairs of nodes in the graph, then finding the shortest weighted path between pairs of nodes in our priority node list. While generating the all pairs shortest paths would be reasonable to execute, finding the shortest path of our priority nodes would again be very time inefficient. It would require finding all permutations of the priority nodes, calculating the path cost for each, and finding the minimum cost path among those. This would have a time complexity of O(n^3 + m!), where n = number of nodes in the graph and m = number of priority nodes.

As we continued to contemplate our problem, we recognized that it was very close in nature to the Traveling Salesman Problem (TSP) that we learned about in Module 11. Although we were not attempting to generate a Hamiltonian Cycle, we were seeking the minimum cost path that would take us through a set of nodes in a graph- essentially a TSP tour of a certain subset of the graph, without returning to the starting node. Since TSP is in NP Complete, there is no way to generate an optimal output in polynomial time. This led us to conclude that the best method to find our ideal bus route would be to utilize the methods learned in Module 12 and generate an approximation algorithm for a modified TSP tour.

Results and Analysis

We were able to again to utilize the existing methods in the Networkx package and generate an approximate minimum cost path through all of our priority nodes. The approximation we utilized was a Metric Approximation, implemented with Christofides algorithm.

The Christofides algorithm begins by creating a minimum spanning tree (M) for the desired nodes. A set is created of odd-degree vertices (oV) from M. oV contains, at a minimum, all the leaf nodes of M and by the handshaking lemma, oV has an even number of vertices. A minimum weight perfect matching is made from the complete graph of oV. The edges of that matching are combined with the edges of M and a graph is formed with all nodes having an even degree. A Eulerian tour is made of the new graph. Duplicate nodes are removed from the path, and an approximate minimum cost tour is generated[6].

Christofides algorithm, like other metric TSP algorithms, utilizes the triangle inequality to prove the validity of the tour that is generated. That is, for any 3 nodes in a weighted, undirected graph G(V,E) with non-negative with edges:

$$(u, v), (v, w), (u, w)$$

$$c(u, w) <= c(u, v) + c(v, w)$$

Since our graph is undirected, has non-negative weight edges, and is built on an X,Y coordinate plane, the triangle inequality does hold and the tour generated by removing duplicate nodes form the Eulerian tour is valid. Since the algorithm utilizes a minimum spanning tree, we know that each edge added to that tree is the minimum valued edge that connects the existing tree (starting from an empty tree) to the remaining nodes that are not yet in the tree. Such edges are added until no nodes remain unconnected. So we know that the tree that is generated both spans all nodes of our subgraph that includes priority nodes, and is a set of minimum valued edges spanning those nodes. So Christofides algorithm has generated an approximate shortest path between all our priority nodes.

When considering the execution of Metric TSP algorithms, we know that the cost of the MST generated by the approximation algorithm will always be less than or equal to the cost c of the

optimal solution (OPT) of a graph G, as a MST can be generated by deleting any edge from a tour, and all edges are non- negative.

$$c(MST) <= c(OPT)$$

Since we are using a Eulerian tour (W) to traverse the MST, we will visit every edge/vertex exactly twice. So the cost for this walk will be equal to two times the cost of the tree itself.

$$c(W) = 2c(MST)$$

Combining these two factors, we know that:

$$c(W) <= 2 * c(OPT)$$

Since we know the triangle equality holds, the path that is generated when duplicate nodes are removed from W must have a value less than W, so for our final resulting path P

$$c(P) <= c(W) <= 2 * c(OPT) \rightarrow c(P) <= 2 * c(OPT)$$

The Christofides further improves this approximation.

Let us define:

P to be the perfect matching of the complete graph of oV

$N^{'}$ be the OPT TSP tour of oV

$R_1,\ R_2$ to be two separate perfect matchings of the G(oV) on the edges of $N^{'}$ taken independently.

Then:

$$c(P) = min\ c(R_1, R_2)$$

as stated above

$$c(P) \leq\ c(N^{'})/2$$

as the minium of R_1 and R_2 will be at most the average of $N^{'}$

$$c(P) \leq\ c(OPT)/2$$

as oV is a subset of G(V) (and thus smaller than G(V)), the optimal tour of oV must be less than $c(OPT)$, and $N^{'}$ is the optimal tour of oV.

When we combine the edges of P with the edges of the MST previously generated we have:

$$c(P) \leq\ c(OPT)/2$$

$c(MST) \leq c(OPT)$ (as shown in the Metric TSP proof)

$$c(P) + c(MST) \leq\ 3c(OPT)/2$$

The Christofides algorithm has generated a 3/2 approximation of the optimal TSP tour. Since our desired "tour" is truly a path, there is one less edge than in a true TSP tour and is thus less costly. Clearly the algorithm will also yield at least a 3/2 approximation of our path.

Link to relevant source file

Approximate Shortest Path Including All Priority Locations

Conclusion

In answering the question: `Given Portland roads as the network we travel on. Can we determine optimal locations for a given number of bus stops, such that we are minimizing distance to amenities?` we successfully implemented the algorithms and modules we learned in this Algorithms course. This project provides an interesting analysis of the efficacy of Portland's current bus stop locations. What we found is that GPMetro's bus stop locations and routes are quite similar to the Traveling Salesman path, with a few interesting notes:

- GPMetro busses travel on major roads and tend not to take detours. This makes sense as Portland is generally quite walkable so small detours may not be helpful and increase the length of the route.
- The areas our path reached are covered by the various bus routes with a few exceptions.

- A few of the Portland bus routes travel in a loop, which provides access to more residential areas that the Traveling Salemen Path misses.

A drawback to our process was that the Graph is quite large with some of the roads being redundant (as in 2 short roads side by side could somehow be combined). This made it difficult to explore algorithms that have worse time complexity, such as Floyd Warshall and opt for an approximation algorithm. We also only included amenities in our data, but for a more realistic analysis we may want to consider other locations:

- high population areas
- low parking areas
- other desirable locations such as parks, restaurants, offices etc.

This could provide a more real-life analysis, exploring questions like: What single route might residents be taking using Dijkstra's.

Our analysis shows that it is possible to generate an approximate ideal bus route utilizing graph algorithms. Due to the complexity of the problem we were not able to generate a true optimal path, but rather 3/2 approximation. We visually compared our path with the existing bus routes in Portland:

While there are some similarities, there are substantial differences as well.

It is likely the Metro routes consider additional factors we did not address with this project. Future expansion of the project could include utilizing census data to factor population density into the route, updating the graph generated from the roads data to reflect road direction- some roads are not bidirectional, and partitioning our single path into multiple shorter paths more akin to the multiple routes utilized by the Metro system.

Shuiming chen:

After we finished this project, I have a more in-depth understanding about dijkstra's algorithm, floyd warshall and TSP. We have learnt these in class but mostly focus on the theorem part, but this time the project helps me to really use these in practical scenarios though we haven't

applied them all. I mainly focus on the front part of this project, including in the beginning of applying dijkstra's algorithm as our main algorithm, etc.

Amanda:

I learned quite a bit from this project. I was excited to learn of the availability of GIS data for the road systems in Maine. I think I will definitely utilize that for future projects. This project taught me a lot about applying theories learned in class to real world problems. Particularly, I learned about the challenges that come from trying to adapt and utilize a textbook solution when your problem is not textbook perfect.

Emma:

There are a couple things I learned from this project:

- working on an adapting problem - exploring different algorithms with real data and contending with related drawbacks that adjust the solution.
- wrangling Geo data quickly and using packages like shapely, geopandas and NetworkX.
- when working with Geo data plotting on a basemap regularily is very important to verify each step in the code!

This was a very valuable exercise from me as I'd like to continue doing projects like this one to learn more about the city I live in. I think this was a needed sample of the work I'll be doing next semester in PPUA 5262 Big Data for Cities.

References

1. Huitt, W. (2007). Maslow's hierarchy of needs. Educational Psychology Interactive. Valdosta, GA: Valdosta State University. Retrieved Nov 13, 2023 from, http://www.edpsycinteractive.org/topics/regsys/maslow.html
2. Greater Portland Transit District. (2023). 2023 Operating Budget. Retrieved Nov 13, 2023, from https://gpmetro.org/DocumentCenter/View/1497/2023-Operating-Budget--Approved-22323?bidId=
3. Maine Department of Transportation. (n.d.). MaineDOT Public Roads. ArcGIS Hub. Retrieved December 13, 2023, from https://maine.hub.arcgis.com/datasets/maine::mainedot-public-roads/about
4. Google Maps. (n.d.). [Portland Bus Root Locations]. Retrieved December 1, 2023, from https://www.google.com/maps/d/edit?mid=1ndsALekiokpddnr-6D7I4t2-j3xU9xs&ll=43.67155787430646%2C-70.2770516&z=13
5. Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008
6. N. Christofides, Worst-case analysis of a new heuristic for the travelling salesman problem, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

Appendix

Code

Link to project GitHub repo

```
Filename: extract_data.py


def get_gis_data():
    """
    This function fetches data from the Maine GIS website and returns
a dataframe
    """

    # Base url to fetch data
    base_url =
"https://gis.maine.gov/arcgis/rest/services/dot/MaineDOT_OpenData/Map
Server/52/query"

    # Query parameters
    params = {
        'where': '1=1',
        'outFields': '*',
        'outSR': 3857,
        'f': 'json'
    }

    # initialize variables for paginating through data
    batch_size = 1000
    offset = 0

    # initialize lists to store data
    attributes = []
    geometry = []

    # iterate fetching batches of data, 1000 at a time until there is
not more data left
    while True:
        params['resultOffset'] = offset
        response = requests.get(base_url, params=params)

        if response.status_code == 200:

            data = response.json()

            # Extract features from the response and add them to the
result
            features = data.get('features', [])
            attributes.extend([feature.get("attributes", {}) for
feature in features])
            geometry.extend([feature.get("geometry", {}) for feature
in features])

            # break out of loop if this is the last batch of data
```

```
            if len(features) < batch_size:
                break

            offset += batch_size

        else:

            print(f"Failed to retrieve data. Status code:
{response.status_code}")
            break

    # create dataframe with attributes and geometry data
    df = pd.DataFrame(attributes)
    df['geometry'] = geometry

    return df

def convert_to_line_string(geo):
    """
    This function converts the geometry column into a shapely
LineString
    """
    path = [tuple(point) for point in geo['paths'][0]]
    return LineString(path)


# run main
if __name__ == '__main__':
    # get data
    print('Fetching data...')
    data = get_gis_data()
    portland_df = data[data['townname'] == 'Portland']

    # convert geometry column into shapely format so that we can use
geopandas
    print('Converting data to geopandas dataframe...')
    portland_df_copy = portland_df.copy()

    # convert geometry column into shapely LineString
    print('Converting geometry column to LineString...')
    portland_df_copy['geometry'] =
portland_df_copy['geometry'].apply(convert_to_line_string)

    # create geopandas dataframe
    print('Creating geopandas dataframe...')
    portland_gdf = gpd.GeoDataFrame(portland_df_copy,
geometry='geometry')

    # save geopandas dataframe to file
    print('Saving geopandas dataframe to file...')
    portland_gdf.to_file('data/portland_roads.geojson',
driver='GeoJSON', crs='epsg:3857')

    ax = portland_gdf.plot(aspect=1, figsize=(60, 30), color="k")
    ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik,
```

```
zoom=15, crs=portland_gdf.crs)
    plt.savefig('figs/portland_roads.png')
```

```
Filename: build_intersections.py


# read in data from geojson and plot with a basemap
gdf = gpd.read_file('data/portland_roads.geojson')

def plot(data, name):
    """
    plot data with a basemap
    """
    ax = data.plot(aspect=1, figsize=(60, 30), color="k",
linewidth=5, markersize=1)
    ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik,
zoom=15, crs=data.crs)
    plt.suptitle(f'Portland Roads: {name}', fontsize=20)
    plt.savefig(f'figs/{name}.png', bbox_inches='tight')

congress = gdf[gdf['strtname'] == 'CONGRESS ST']
plot(congress, "congress")

def get_intersections(data):
    """
    build and return intersection distionary
    # key: tuple of intersection coordinates
    # value: list of indexes of roads that intersect at that
intersection
    """
    intersections = {}
    for i in range(len(data)):
        # get coordinates of road
        road = data.iloc[i]

        first_coord = road["geometry"].coords[0]
        last_coord = road["geometry"].coords[-1]

        first_coord = (int(first_coord[0]), int(first_coord[1]))
        last_coord = (int(last_coord[0]), int(last_coord[1]))

        # add intersection to dictionary
        if (first_coord[0], first_coord[1]) not in intersections:
            intersections[(first_coord[0], first_coord[1])] =
{"roads": [road["OBJECTID"]], "neighbors": [(last_coord[0],
last_coord[1])]}
        else:
            intersections[(first_coord[0], first_coord[1])]
["roads"].append(road["OBJECTID"])
            intersections[(first_coord[0], first_coord[1])]
["neighbors"].append((last_coord[0], last_coord[1]))

        if (last_coord[0], last_coord[1]) not in intersections:
```

```
            intersections[(last_coord[0], last_coord[1])] = {"roads":
[road["OBJECTID"]], "neighbors": [(first_coord[0], first_coord[1])]}
        else:
            intersections[(last_coord[0], last_coord[1])]
["roads"].append(road["OBJECTID"])
            intersections[(last_coord[0], last_coord[1])]
["neighbors"].append((first_coord[0], first_coord[1]))

    return intersections

def get_dataframe(intersections):
    """
    build and return dataframe of intersections
    # index: tuple of intersection coordinates
    # columns: list of indexes of roads that intersect at that
intersection
    # values: number of roads that intersect at that intersection
    """
    df = pd.DataFrame(index=intersections.keys(), columns=
['intersection_coords', 'neighbors', 'road_indexes', 'num_roads'])
    for key in intersections:
        df.at[key, 'intersection_coords'] = key
        df.at[key, 'neighbors'] = intersections[key]["neighbors"]
        df.at[key, 'road_indexes'] = intersections[key]["roads"]
        df.at[key, 'num_roads'] = len(intersections[key]["roads"])
    return df

intersections = get_intersections(gdf)
df = get_dataframe(intersections)
sorted_intersections = df.sort_values(by=['num_roads'],
ascending=False)

# get top 10 intersections
top_10_intersections = sorted_intersections.head(10)
print(top_10_intersections)
indexes = []
for i in range(len(top_10_intersections)):
    intersection = top_10_intersections.iloc[i]
    indexes += intersection['road_indexes']

# filter original dataframe to only include roads that intersect at
first intersection
first_10_intersection_roads_df = gdf[gdf['OBJECTID'].isin(indexes)]

# plot roads that intersect at first intersection
plot(first_10_intersection_roads_df, "first_10_intersection_roads")

sorted_intersections.reset_index(drop=True, inplace=True)
# convert to geodataframe
sorted_intersections['geometry'] =
sorted_intersections['intersection_coords'].apply(Point)
sorted_intersections = gpd.GeoDataFrame(sorted_intersections,
geometry='geometry')
# convert fields to string
sorted_intersections['intersection_coords'] =
```

```
sorted_intersections['intersection_coords'].astype(str)
sorted_intersections['road_indexes'] =
sorted_intersections['road_indexes'].astype(str)
sorted_intersections['neighbors'] =
sorted_intersections['neighbors'].astype(str)
# write to geojson
sorted_intersections.to_file('data/intersections.geojson',
driver='GeoJSON', crs='epsg:3857')

# plot intersections
plot(sorted_intersections, "sorted_intersections")
```

```
Filname: utils.py


def read_data():
    """
    Returns a geodataframe of all the priority locations, a
geodataframe of all the intersections, and a geodataframe of all the
roads
    """
    groc = pd.read_csv('data/Portland Bus Route Locations- Grocery
Stores.csv')
    scho = pd.read_csv('data/Portland Bus Route Locations-
Schools.csv')
    hc = pd.read_csv('data/Portland Bus Route Locations- Healthcare
Facilities.csv')
    all = pd.concat([groc,scho,hc])
    all['WKT'] = gpd.GeoSeries.from_wkt(all['WKT'])
    all_gdf = gpd.GeoDataFrame(all, geometry =
'WKT').set_crs("EPSG:4326")
    all_gdf = all_gdf.to_crs("EPSG:3857")

    intersections = gpd.read_file('data/intersections.geojson')
    # set geometry to the geometry column
    intersections = intersections.set_geometry('geometry')

    roads = gpd.read_file('data/portland_roads.geojson')

    return all_gdf, intersections, roads

def load_graph(roads):
    """
    Returns a graph of the roads
    """
    G = nx.Graph()
    # populate edges as roads with their length as weight
    for i in range(len(roads)):
        # get coordinates of road
        road = roads.iloc[i]
        length = road["Shape_Length"]
        # add road as edge to graph, having its start and end
coordinates as the intersection nodes
```

```
        first_coord = road["geometry"].coords[0]
        last_coord = road["geometry"].coords[-1]
        # only add if its not a road to itself
        if first_coord != last_coord:
            # casting to int to approximate coordinates as nodes
            G.add_edge((int(first_coord[0]), int(first_coord[1])),
(int(last_coord[0]), int(last_coord[1])), weight=length)

    return G

def get_largest_component_as_graph(G):
    """
    Returns the largest connected component of the graph
    """
    # get largest connected component
    largest_component = max(nx.connected_components(G), key=len)
    # create subgraph of largest connected component
    G = G.subgraph(largest_component)
    return G
```

```
Filename: shortest_path.py


# import data
all_gdf, intersections, roads = read_data()

def gen_plot():
    """
    plot priority locations data with a basemap
    """
    ax = all_gdf.plot(figsize=(10, 8), column = 'description',
categorical=True,legend=True)
    ax.axis('off')
    ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik,
zoom=15, crs=all_gdf.crs)
    plt.suptitle('Portland Priority Locations', fontsize=10)
    plt.savefig('figs/priority_locations.png')

gen_plot()

G = load_graph(roads)

# loop intersections and all_gdf and add columns for distance from
each node to points of interest
# store distance
for i in range(len(all_gdf)):
    point = all_gdf.iloc[i,0]
    intersections[all_gdf.iloc[i,1]] =
intersections['geometry'].distance(point)

print(intersections.head())

# extract minimum distances and create list of priority nodes
```

```
loc_list = []
for i in range(4,36):
    min_node = intersections.iloc[:,i].idxmin()
    tup = (intersections.iloc[min_node].geometry.x
,intersections.iloc[min_node].geometry.y)
    loc_list.append(tup)

print(len(loc_list))
print(loc_list[0])

# using simple path does not exectute d/t running time
shortest_paths = nx.shortest_path(G, source = loc_list[0],
weight='weight')
shortest_path_cost = nx.shortest_path_length(G, source = loc_list[0],
weight='weight')

print(f"Shortest path from {loc_list[0]} to {loc_list[1]}:")
print(shortest_paths[loc_list[1]])
print(shortest_path_cost[loc_list[1]])

for node in loc_list:
    path = shortest_paths[node]
    if set(loc_list).issubset(path):
        print (shortest_paths[node])


# find the shortest path between the two points
tsp = nx.approximation.traveling_salesman_problem
path = tsp(G, nodes=loc_list, cycle=False)
sub_graph = G.subgraph(path)
locations = {loc: (Point(loc).x, Point(loc).y) for loc in path}

fig, ax = plt.subplots(figsize=(50, 50))
nx.draw(
    sub_graph,
    locations,
    ax = ax,
    node_size=60,
    width=8,
    node_color="k",
    edge_color="k",
    alpha=0.8,
)
ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik,
zoom=15, crs=intersections.crs)
plt.savefig(f'figs/shortest_path.png', bbox_inches='tight')
```