GP_1_03_knopp_henselx2_group_project_3

December 6, 2023

1 Group Project #3

This is a group activity and worth 100 points. Each team should consist of 2 or 3 members - working alone or in a group withbers is not permitted more than 3 mem. You can create a new group or retain your previous team members. The choice is yours. Use Canvas to create your own group. Create a code/name group (max 10 letters).

Team Members: (Name and Identikey)

- Member 1: Daniel Knopp
- Member 2: Kenzie Hensel
- Member 3: Ryan Hensel
- Group Name: Group 1

Due Date: Dec 7th 12:10 pm (Canvas)

Delivery include * Report, Code and a Presentation

Upload: PDF, Jupyter Notebook file, and a PPT for the oral presentation.

2 (100 points) Find the shortest route in your city between two points of interest.

Select a city and plan your route with a map and knowledge of traffic. You need to evaluate a shortes route (in time) from a designated initial (A) to a final destiny (J).

For a given scenario, you must evaluate the movement between points A and J in both directions. The time required to move from A to J and vice versa is the same. To assist in this evaluation, you must define internal nodes B, C, D, E, F, G, H, and I in the map provided as a reference. The number of internal nodes must be between 6 to 10 (inclusive). The connectivity of internal nodes will vary based on the specific case presented.

To solve a problem that requires finding the shortest path between two points in a graph, you need to implement Dijkstra's Algorithm. You have the choice to use any data structure for representing the graph: edge list, adjacency list, adjacency matrix, etc. However, it's important to maintain consistency throughout. The weight of each connection in the graph should represent the time taken to traverse it and can be a float/decimal value.

Your team is required to develop two scenarios considering uncertainties. The first scenario is the base case with an average time. The second scenario is the rush hour with high traffic, which causes delays on most routes (not all of them). Your evaluation of both scenarios should incorporate some level of uncertainty. This means that a route can be blocked or delayed by 10 times or more. To make it more realistic, you need to identify at least three pairs of nodes that may encounter such a situation.

Tips: * For the base case scenario, use Google Maps or Apple Maps to define the average time from each pair of nodes (weight). * For the rush hour scenario, apply a factor between 1 -> 3. For example, you can define a random number using a normal distribution with media of 2 and a standard deviation of 0.5. Evaluate each case (node1->Node2) and apply your own base modification. Use the random library to solve this impact. * Add uncertainty is a similar model for rush hour but with high impact and low probability events. Each case can be a discrete distribution (0,1) with 80% nothing happened with a 20% this unlike event will happen.
* Every simulation can be different, and this is ok. You will present one case. * If you have asspumtions explain in your report.

Provide context and show the solution using a map of the selected city. Below is a map you can use as a reference.

You need to create a report that includes: * Description of your approach * Description of the Data Structure used and the algorithm implementation. * Test your algorithm with a manual calculation for the base case scenario. * Show the results for each case scenario. Simulation base and rush hour scenarios * Discuss your results.

Prepare a PowerPoint presentation with 3-6 slides. Use your Jupyter notebook/Python code to run your solution in real time. All team members must be present to be evaluated, however one or more can be the preesent during that day. Presentation will start on Dec 7th, and, if needed it, it can be continued on Dec 12th.

Report (70 points), Presentation (20), PeerReview evaluation/Audience (10)

Note: The report can be a WORD/PDF file or a Jupuyter Notebook file.

Description of Approach

The first thing we needed to create for this project is a Node class that will be the template for each of the nodes in the graph. For this class, we need to keep track of useful information that is necessary for implementing Dijkstra's Algorithm. These attributes are: numerical ID, location name (from Google Maps), a list of neighbors, shortest time to get to this node from start node, has the node been visited, and previous node that arrived here for the latest minimum time. For the list of neighbors, we chose to have each neighbor object be itself also a list - this way we can keep track of path information between neighbors without having to define seperate edge lists. The individual neighbor objects are lists that contain the following information: node object, baseline time between nodes, and modified time between nodes (modifications include rush hour scaling and unexpected events/uncertainties). We also created a fucntion to plot the graph visually so that we could overlay it on top of a Google Maps screenshot, so for that to work well we stored pixel-coordinates in each graph node as well that represents the pixel coordinates of the waypoint on the Google Maps screenshot we used.

We created a number of helper functions to save space when writing Dijkstra's Algorithm. These helper functions are: function to apply rush hour time scaling, function to apply unexpected event/uncertainty scaling, function to output final graph data (after Dijkstra's algorithm has been executed) as a table, and a function to visualize the final graph data (after Dijkstra's algorithm has been executed) as a web of nodes (in the PPT we overlayed these webs onto the Google Maps screenshot).

Finally we created a function to execute Dijkstra's Algorithm on the graph nodes. We needed to choose a starting and ending node, we chose Node 1 and 3 because we thought these provided the most interesting route based on the points of interest we chose in Google Maps. The way we coded the algorithm is as follows:

Initialize a to_visit list and add the starting node to the list of nodes to visit

Continue looping until the to_visit list is empty

Sort the to_visit list by order of closest nodes to the starting node (min time value at node)

Take the next node out of the to_visit list and mark it as visited

Get the information about all the node's neighbors

Look at moving from current node to neighbor, apply any modifiers to the baseline travel time (rush hour, uncertainty) and adjust the stored travel time for both directions (once it's changed the first time, will never be changed again during the algorithm, see conditional statements in code below)

For any neighbor who's stored minimum time is greater than the current node's time plus the travel time between the nodes, overwrite the neighbor node's minimum travel time and store the current node as the neighbor node's predecessor

Add the neighbor nodes to the to_visit list(making sure not to not add the neighbor node to the to_visit list if it has already been visited or is already in the to_visit list)

After have finished looping over all nodes (to_visit list is empty), we get the minimum travel time stored in the end node and get the optimal path by following the path of predecessors back from the ending node to the starting node (optimal path is the reverse of this order)

See the code below for specific details on each line of the algorithm in the code block comments. Note that what I am referring to as the 'to_visit list' above is actually a FIFO Queue, explaination for why this was done is in blue boxes below.

```
[ ]: # Formatting for HTML export so that code blocks are not truncated, but instead␣
      ↪a horizontal scroll bar is added
     from IPython.display import display, HTML
     display(HTML('<style>pre { white-space: pre; overflow-x: scroll; }</style>'))
```

```
<IPython.core.display.HTML object>
```

```
[ ]: # Define a class for each node in the graph
     class Node:

         # Constructor
```

```python
    def __init__(self, id, name, pixel_coords):
        self.id           = id
        self.name         = name
        self.neighbors    = []
        self.time         = 9999
        self.visited      = False
        self.predecessor  = None
        self.pixel_coords = pixel_coords

    # Add a neighbor to the node
    def add_neighbor(self, neighbor, travel_time_baseline,␣
↪travel_time_modified):
        self.neighbors.append([neighbor, travel_time_baseline,␣
↪travel_time_modified])

    # Get the neighbors of the node
    def get_neighbors(self):
        return self.neighbors

    # Get the name of the node
    def get_name(self):
        return self.name

    # Get the id of the node
    def get_id(self):
        return self.id

    # Get the id of the node
    def get_name_with_id(self):
        return f'({self.id})_{self.name}'

    # Get the min travel time of the node
    def get_time(self):
        return self.time

    # Set the min travel time of the node
    def set_time(self, time):
        self.time = time

    # Get the visited status of the node
    def get_visited(self):
        return self.visited

    # Set the visited status of the node
    def set_visited(self, visited):
        self.visited = visited
    # Get the pixel coordinates of the node
    def get_pixel_coords(self):
        return self.pixel_coords

    # Get the predecessor of the node
    def get_predecessor(self):
        return self.predecessor

    # Set the predecessor of the node
    def set_predecessor(self, predecessor):
        self.predecessor = predecessor

    # Method to reset modified travel times back to default (-1), reset visited␣
 ↪status, reset time to high number, reset predecessor
    def reset_node(self):
        self.time = 9999
        self.visited = False
        self.predecessor = None
        for neighbor in self.neighbors:
            neighbor[2] = -1

    # Print the node - behavior is the same as get_id()
    def __str__(self):
        return f'({self.id})'
```

The code below defines a helper function for scaling travel time between a pair of nodes based on uncertainty (20% chance of an event occuring, where the event randomly scales the travel time by a factor of between 10 and 15x). This conceptually represents things such as accidents and unexpected road construction, for example.

[ ]:
```
import random

# Define a function to scale travel time value based on occurance of uncertain␣
 ↪events (i.e. accidents, traffic light outages, etc.)
def uncertainty_scaling(time):

    # Check if the unlikely event will occur (20% chance)
    if random.random() <= 0.2:

        # Scale the travel time by a random number between 10 and 15
        return round(time * random.randint(10, 15), 1)

    # Otherwise the event did not occur, return the original travel time
    return time
```

The code below defines a helper function for scaling travel time between a pair of nodes based on rush hour traffic (normal distribution function with mean at 2 and a standard deviation of 0.5). To model this distribution, it was easiest to import the norm function from the scipy package instead of modeling the distribution with fundamental equations.

```
[ ]: from scipy.stats import norm

     # Define a function to scale travel time value during rush hour
     def rush_hour_scaling(time):

         # Define a normal distribution with a mean of 2 and a standard deviation of␣
      ↪0.5 for the scaling factor
         scale_dist = norm(loc=2, scale=0.5)

         # Get a random number from the normal distribution
         scale_factor = scale_dist.rvs()

         # Scale the travel time by the random number
         return round(time * scale_factor, 1)
```

The code below defines a helper function for sorting the nodes in the queue such that the next closest node is always in front. This requirement was not discussed in class, so we started using a Queue as a data structure in the algorithm. Once we learned by looking online that we need to always go to the next closest node, we just decided to implement a function to sort the queue (kind of defeating the purpose of using a queue now…) instead of rewriting our algorithm to use a list instead of a queue (it would make more sense to just use a list and sort the list each time we are about to move to a new node since sorting a queue requires getting all the data out, sorting it, and then building a new queue).

```
[ ]: import queue

     # Create a function that sorts the queue based on the time attribute of the node
     def sort_queue(unsorted_queue):
         sorted_list = sorted(unsorted_queue.queue, key=lambda node: node.get_time())
         out = queue.Queue()
         for node in sorted_list: out.put(node)
         return out
```

The code below defines a helper function for visualizing the node data and all neighboring connecitons in a tabular format - useful for debugging and verifying results.

```
[ ]: import pandas as pd

     # Function to print the graph data in tabular format
     def print_table(start_node, node_list):

         # Create an empty DataFrame
         df = pd.DataFrame()

         # Loop through each node and add data to the pandas DataFrame
         for node in node_list:
```

```
        # Initialize a variable to store the optimal path from start to end node
        path = []

        # Loop from the end node until reaching the starting node by following␣
↪predecessors
        cur = node
        while cur != None:

            # Add the current node to the path
            path.append(str(cur))

            # Get the predecessor of the current node
            cur = cur.get_predecessor()

        # Reverse the path (since we started at the end node and went backward␣
↪in the looping structure above)
        path.reverse()

        # If the path is just the current node, then there is no optimal path␣
↪to the node
        if path == [str(node)]:
            path = ['None']

        # Create a dictionary to store the data for the current node
        data = {
            'Current Node': node.get_name_with_id(),
            't_min': node.get_time(),
            'Visited?': 'Yes' if node.get_visited() == 1 else 'No',
            'Optimal Path from ' + str(start_node): "[" + ", ".join([pre for␣
↪pre in path]) + "]"
        }

        # Get the neighbors and add their data to the dictionary
        neighbors = node.get_neighbors()
        for i in range(5):
            try:
                data['Neighbor Node 0' + str(i+1)] = neighbors[i][0].
↪get_name_with_id()
                data['t_base_0'        + str(i+1)] = neighbors[i][1]
                data['t_mod_0'         + str(i+1)] = neighbors[i][2]
            except:
                data['Neighbor Node 0' + str(i+1)] = '-'
                data['t_base_0'        + str(i+1)] = '-'
                data['t_mod_0'         + str(i+1)] = '-'

        # Append the data to the pandas DataFrame
        df = pd.concat([df, pd.DataFrame(data, index=[0])])

    # Reset the index of the DataFrame and drop the default first column (row␣
↪index)
    df = df.reset_index(drop=True)

    # Set the properties of the DataFrame to align the text to the left
    styles = [
        dict(selector="th", props=[("text-align", "left"), ("white-space",␣
↪"nowrap")]),
        dict(selector=""  , props=[("text-align", "left"), ("white-space",␣
↪"nowrap")])
    ]

    # Set the table content to be left aligned and set the numerical precision␣
↪to 1 decimal place
    df = df.style.set_properties(**{'text-align': 'left'}).
↪set_table_styles(styles).format(precision=1)

    # Display the pandas DataFrame
    display(df)
```

The code below is our implementation of Dijkstra's Algorithm. See methodology discussion above for overview of the procedure. See comments throughout code below for specific details of each piece of the code. For this assignment, we originally chose to use a queue data structure to track the next nodes we needed to visit. This was because in class we were told that it did not matter the order we visited the neighbor nodes but that it only mattered that we visit the 1st neighbors before the 2nd neighbors or 3rd neghbors, relative to the starting node. This turned out to be incorrect information as we ran into examples where we were not getting the correct answer using this methodology. We then learned from searching online that the order you visit each neighbor is crucial to Dijakstra's algorithm. We, in fact, need to always visit the next closest (node with the minmum time value) neighbor who is adjacent to any of the nodes we have already visited. The way we implemented this is we just created a helper function which sorts the queue before it tries to get the next node to visit. As discussed in the comment above the helper function earlier in this notebook, this defeats the purpose of using a queue but we chose this workaround insted of rewriting the algorithm given the new information we learned from searching online. For the queue implementation itself, we chose to import the queue package for creating a first-in-first-out queue instead of creating a special class for creating a queue. This was done out of convenience and simplicity of the notebook since we have already hand-coded implementations of queues in previous assignments in this class. Of course, given the new information we were not given in-class, it would have made the most sense to just use a list to keep track of nodes to visit, and then sort that list before poping off the next node to visit.

We have also setup the function to run in 'verbose' mode where if that flag is set to True (default if False), then detailed logging messages about each step in the algorithm are printed for the user to understand and confirm the algorithm is working correctly.

```
[ ]: # Define a function to implement Dijkstra's algorithm
     def find_min_travel_time(start_node, end_node, is_rush_hour,␣
      ↪include_uncertainty, node_list, verbose=False):

         if verbose: print(f'Starting Dijkstra\'s Algorithm: {start_node} ->␣
      ↪{end_node}, Rush Hour: {is_rush_hour}, Uncertainty: {include_uncertainty}',␣
      ↪end='\n\n')

         # Create a queue of nodes we need to visit
         to_visit = queue.Queue()

         # Add the starting node to the queue
         to_visit.put(start_node)
         if verbose: print(f'Added {start_node} to queue: Front <- {" | ".
      ↪join([str(node) for node in to_visit.queue])} <- Rear', end='\n\n')

         # While there are still nodes to visit
         while not to_visit.empty():

             # If verbose, print the graph table
             if verbose:
                 print('Current Status of Graph (table format):')
                 print_table(start_node, node_list)
                 print()

             # Sort the queue by the time attribute of the node
             to_visit = sort_queue(to_visit)
             if verbose: print(f'Sorting queue by time attribute of node (must visit␣
      ↪next closest node first): Front <- {" | ".join([str(node) for node in␣
      ↪to_visit.queue])} <- Rear', end='\n\n')

             # Get the next node to visit and set as visited
             current_node = to_visit.get()
             current_node.set_visited(True)
             if verbose: print(f'Visiting node: {current_node}, popped the node from␣
      ↪queue: Front <- {" | ".join([str(node) for node in to_visit.queue])} <-␣
      ↪Rear', end='\n\n')

             # Get the neighbors of the current node
             neighbors = current_node.get_neighbors()
             if verbose: print(f'Getting neighbors: [{", ".join([neighbor[0].
      ↪get_name_with_id() for neighbor in neighbors])}]', end='\n\n')

             # Loop over each neighbor of the current node
             for neighbor in neighbors:

                 # Get the neighbor node object from neighbor node list element
```

```
            neighbor_node = neighbor[0]
            if verbose: print(f'Checking neighbor: {neighbor_node}', end='\n\n')

            # Get the (potentially modified) travel time between current node␣
↪and neighbor node
            neighbor_time = neighbor[2]

            # If the travel time has not yet been modifed
            if neighbor_time < 0: # Note in code snippets below this algorithm␣
↪that the initial value of this element for each node is -1

                # Get the baseline travel time (this element of the list will␣
↪never be changed, even in rush hour and uncertainty scenarios)
                neighbor_time = neighbor[1]

                # If it is currently rush hour
                if is_rush_hour:

                    # Scale the travel time between current and neighbor node,␣
↪update the local travel time variable
                    pre_scale_time = neighbor_time
                    neighbor_time  = rush_hour_scaling(neighbor_time)
                    if verbose: print(f'Route experiences rush hour, scaling␣
↪factor of {neighbor_time / pre_scale_time} applied', end='\n\n')

                # If we are including uncertainty events (don't want to include␣
↪for the baseline visualization)
                if include_uncertainty:

                    # Apply the scaling for any unexpected events
                    pre_scale_time = neighbor_time
                    neighbor_time  = uncertainty_scaling(neighbor_time)
                    if verbose: print(f'Route experiences uncertainty, scaling␣
↪factor of {neighbor_time / pre_scale_time} applied', end='\n\n')

                # Update the modified neighbor time element stored in the␣
↪neighbor node list object
                neighbor[2] = neighbor_time
                if verbose: print(f'Travel time from {current_node} to␣
↪{neighbor_node} updated to {neighbor_time} minutes', end='\n\n')

                # Find and update the other direction of this edge/node␣
↪connection by finding the neighbor node in the neighbor node's neighbor list␣
↪that matches the current node
                for neighbor_neighbor in neighbor_node.get_neighbors():
                    # If the neighbor's neighbor is the current node
                    if neighbor_neighbor[0] == current_node:

                        # Update the modified travel time element stored in the␣
↪neighbor's neighbor node list object
                        neighbor_neighbor[2] = neighbor_time
                        if verbose: print(f'Travel time from {neighbor_node} to␣
↪{current_node} updated to {neighbor_time} minutes', end='\n\n')

            # Compute the travel time from current node to neighbor node,␣
↪including min time to get to current node
            travel_time = round(current_node.get_time() + neighbor_time, 1)
            if verbose: print(f'Total travel time from {current_node} to␣
↪{neighbor_node} is {current_node.get_time()} + {neighbor_time} =␣
↪{travel_time} minutes', end='\n\n')

            # Check if the travel time to get to the neighbor node is less than␣
↪the current travel time stored at that neighbor node
            if travel_time < neighbor_node.get_time():
                if verbose: print(f'Travel time of {travel_time} minutes from␣
↪{current_node} to {neighbor_node} is less than previous travel time of␣
↪{neighbor_node.get_time()} minutes at {neighbor_node}, updating␣
↪{neighbor_node} predecessor to {current_node} and min travel time to␣
↪{travel_time} minutes', end='\n\n')

                # Set the travel time to the neighbor node as travel time from␣
↪current node
                neighbor_node.set_time(travel_time)

                # Update the predecessor of the neighbor node to the current␣
↪node
                neighbor_node.set_predecessor(current_node)

            else:

                if verbose: print(f'Travel time of {travel_time} minutes from␣
↪{current_node} to {neighbor_node} is NOT less than previous travel time of␣
↪{neighbor_node.get_time()} minutes at {neighbor_node}, do nothing',␣
↪end='\n\n')

            # If the neighbor node hasn't been visited yet and if it's not␣
↪already in the queue
            if neighbor_node.get_visited() == False and neighbor_node not in␣
↪to_visit.queue:

                # Add the node to the to_visit queue
                to_visit.put(neighbor_node)
                if verbose: print(f'Added {neighbor_node} to queue: Front <- {"␣
↪| ".join([str(node) for node in to_visit.queue])} <- Rear', end='\n\n')

            else:
                if verbose: print(f'{neighbor_node} has already been visited or␣
↪is already in the queue: Front <- {" | ".join([str(node) for node in␣
↪to_visit.queue])} <- Rear', end='\n\n')

            # If verbose logging, print the graph table
            if verbose and neighbor != neighbors[-1]:
                print('Current Status of Graph (table format):')
                print_table(start_node, node_list)
                print()

    # Initialize a variable to store the optimal path from start to end node
    path = []

    # Loop from the end node until reach the starting node by following␣
↪predecessors
    cur = end_node
    while cur != None:

        # Add the current node to the path
        path.append(str(cur))

        # Get the predecessor of the current node
        cur = cur.get_predecessor()

    # Reverse the path (since we started at the end node and went backward in␣
↪the looping structure above)
    path.reverse()

    if verbose:
        print('-----------------------------')
        print('---- Algorithm Complete! ----')
        print('-----------------------------')
        print()

    # Return the optimal path from start to end node and the total travel time
    return end_node.get_time(), path
```

The code below defines a helper function for visually plotting the graph nodes and the travel time between them (including any scaling for rush hour and/or uncertainty, if applicable).

[ ]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

```
def visualize_graph(node_list):

    # Specify the resolution of the reference image (google maps screenshot in␣
  ↪PPT)
    img_res = [1234, 851]

    # Load the background image
    img = plt.imread("google_maps_screenshot.png")

    # Create a figure
    plt.figure(figsize=(img_res[0]/100, img_res[1]/100), facecolor='none')

    # Plot the background image
    plt.imshow(img, extent=[img_res[0]*0.04, img_res[0]*0.94, img_res[1]*0.03,␣
  ↪img_res[1]*0.94])

    # Create a directed graph using networkx
    graph = nx.DiGraph()

    # Add nodes to the graph
    for node in node_list:
        graph.add_node(node.get_id())

    # Add edges to the graph
    for node in node_list:
        for neighbor, _, weight in node.get_neighbors():
            graph.add_edge(node.get_id(), neighbor.get_id(), weight=weight)

    # Manually specify positions (in pixel coordinates relative to the Google␣
  ↪Map screenshot in the PPT) of each node on the graph
    pos = {
        node_list[0].get_id(): (node_list[0].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[0].get_pixel_coords()[1]),
        node_list[1].get_id(): (node_list[1].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[1].get_pixel_coords()[1]),
        node_list[2].get_id(): (node_list[2].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[2].get_pixel_coords()[1]),
        node_list[3].get_id(): (node_list[3].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[3].get_pixel_coords()[1]),
        node_list[4].get_id(): (node_list[4].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[4].get_pixel_coords()[1]),
        node_list[5].get_id(): (node_list[5].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[5].get_pixel_coords()[1]),
        node_list[6].get_id(): (node_list[6].get_pixel_coords()[0], img_res[1]␣
  ↪-node_list[6].get_pixel_coords()[1]),
        node_list[7].get_id(): (node_list[7].get_pixel_coords()[0], img_res[1]␣
↪-node_list[7].get_pixel_coords()[1]),
    }

    # Visualize the graph
    nx.draw(graph, pos, with_labels=True, node_size=1000, node_color='red',␣
↪font_size=12, font_color='white', font_weight='bold', arrowsize=10, width=2.
↪5, edge_color='black')

    # Add the edge weights to the graph
    edge_labels = nx.get_edge_attributes(graph, 'weight')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels,␣
↪font_size=14, font_color='black')

    # Display the graph
    plt.show()
```

The code below is manually inputting the node data and travel times between nodes based on the connections shown in the Google Maps screenshot in the PPT. The travel times between each node were calculated using Google Maps route planning and an average value was computed from the time it takes to go both directions between each pair of nodes.

```
[ ]: # Create all the nodes in the graph, arguments are [id, name, pixel_coords]
node_01 = Node(1, 'Movement_Boulder'        , [ 693, 149])
node_02 = Node(2, 'Mount_Sanitas_Trailhead', [ 128, 330])
node_03 = Node(3, 'Chautauqua_Park'         , [ 342, 738])
node_04 = Node(4, 'Pearl_Street_Mall'       , [ 384, 372])
node_05 = Node(5, 'Fiske_Planetarium'       , [ 606, 655])
node_06 = Node(6, 'Peleton_West'            , [ 840, 433])
node_07 = Node(7, 'Valmont_Dog_Park'        , [1091, 147])
node_08 = Node(8, 'Whole_Foods_Market'      , [ 715, 267])

# Add the neighbors to each node in the format [neighbor_object,␣
↪travel_time_baseline, travel_time_modified]
node_01.add_neighbor(node_04, 7  , -1)
node_04.add_neighbor(node_01, 7  , -1)
node_01.add_neighbor(node_08, 3  , -1)
node_08.add_neighbor(node_01, 3  , -1)
node_01.add_neighbor(node_07, 4  , -1)
node_07.add_neighbor(node_01, 4  , -1)
node_02.add_neighbor(node_03, 7  , -1)
node_03.add_neighbor(node_02, 7  , -1)
node_02.add_neighbor(node_04, 4  , -1)
node_04.add_neighbor(node_02, 4  , -1)
node_03.add_neighbor(node_04, 5  , -1)
node_04.add_neighbor(node_03, 5  , -1)
node_03.add_neighbor(node_05, 4  , -1)
```

```
node_05.add_neighbor(node_03, 4  , -1)
node_04.add_neighbor(node_05, 5  , -1)
node_05.add_neighbor(node_04, 5  , -1)
node_04.add_neighbor(node_08, 5  , -1)
node_08.add_neighbor(node_04, 5  , -1)
node_05.add_neighbor(node_06, 7.5, -1)
node_06.add_neighbor(node_05, 7.5, -1)
node_06.add_neighbor(node_08, 5  , -1)
node_08.add_neighbor(node_06, 5  , -1)
node_07.add_neighbor(node_08, 5  , -1)
node_08.add_neighbor(node_07, 5  , -1)
node_06.add_neighbor(node_07, 5.5, -1)
node_07.add_neighbor(node_06, 5.5, -1)

# Store all the ndoes in a list object for convenience
node_list = [node_01, node_02, node_03, node_04, node_05, node_06, node_07,␣
 ↪node_08]
```

Baseline Results Discussion

The code for testing our baseline case (no rush hour scaling and no uncertainty scaling) is shown below along with the corresponding results. For our base case, our algorithm found the most direct path from starting position 1 to ending position 3 to be path 1,4,3 with a total time of 12 minutes. Upon review and manually calculating the most efficient path by hand to double check, we confirmed this to be true.

```
[ ]: # Reset all the nodes back to initial states
     for node in node_list:
         node.reset_node()

     # Define the starting node, set travel time to 0 (already here), the end node,␣
      ↪and if is rush hour
     start_node          = node_01
     start_node.set_time(0)
     end_node            = node_03
     is_rush_hour        = False
     include_uncertainty = False

     # Find the minimum travel time and path
     min_time, path = find_min_travel_time(start_node, end_node, is_rush_hour,␣
      ↪include_uncertainty, node_list, verbose=True)

     # Print the results
     print(f'The minimum travel time from {start_node.get_name_with_id()} to␣
      ↪{end_node.get_name_with_id()} is {min_time} minutes, following the route␣
      ↪{path}.')

     # Print the final graph data as a table
```

```python
print_table(start_node, node_list)

# Visualize the graph
visualize_graph(node_list)
```

```
Starting Dijkstra's Algorithm: (1) -> (3), Rush Hour: False, Uncertainty: False

Added (1) to queue: Front <- (1) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dffafd0>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (1) <- Rear

Visiting node: (1), popped the node from queue: Front <-  <- Rear

Getting neighbors: [(4)_Pearl_Street_Mall, (8)_Whole_Foods_Market,
(7)_Valmont_Dog_Park]

Checking neighbor: (4)

Travel time from (1) to (4) updated to 7 minutes

Travel time from (4) to (1) updated to 7 minutes

Total travel time from (1) to (4) is 0 + 7 = 7 minutes

Travel time of 7 minutes from (1) to (4) is less than previous travel time of
9999 minutes at (4), updating (4) predecessor to (1) and min travel time to 7
minutes

Added (4) to queue: Front <- (4) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93e09b990>


Checking neighbor: (8)

Travel time from (1) to (8) updated to 3 minutes

Travel time from (8) to (1) updated to 3 minutes

Total travel time from (1) to (8) is 0 + 3 = 3 minutes

Travel time of 3 minutes from (1) to (8) is less than previous travel time of
```

```
9999 minutes at (8), updating (8) predecessor to (1) and min travel time to 3
minutes

Added (8) to queue: Front <- (4) | (8) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dac0b10>


Checking neighbor: (7)

Travel time from (1) to (7) updated to 4 minutes

Travel time from (7) to (1) updated to 4 minutes

Total travel time from (1) to (7) is 0 + 4 = 4 minutes

Travel time of 4 minutes from (1) to (7) is less than previous travel time of
9999 minutes at (7), updating (7) predecessor to (1) and min travel time to 4
minutes

Added (7) to queue: Front <- (4) | (8) | (7) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df95390>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (8) | (7) | (4) <- Rear

Visiting node: (8), popped the node from queue: Front <- (7) | (4) <- Rear

Getting neighbors: [(1)_Movement_Boulder, (4)_Pearl_Street_Mall,
(6)_Peleton_West, (7)_Valmont_Dog_Park]

Checking neighbor: (1)

Total travel time from (8) to (1) is 3 + 3 = 6 minutes

Travel time of 6 minutes from (8) to (1) is NOT less than previous travel time
of 0 minutes at (1), do nothing

(1) has already been visited or is already in the queue: Front <- (7) | (4) <-
Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dfb3c50>
```

```
Checking neighbor: (4)

Travel time from (8) to (4) updated to 5 minutes

Travel time from (4) to (8) updated to 5 minutes

Total travel time from (8) to (4) is 3 + 5 = 8 minutes

Travel time of 8 minutes from (8) to (4) is NOT less than previous travel time
of 7 minutes at (4), do nothing

(4) has already been visited or is already in the queue: Front <- (7) | (4) <-
Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dfca790>


Checking neighbor: (6)

Travel time from (8) to (6) updated to 5 minutes

Travel time from (6) to (8) updated to 5 minutes

Total travel time from (8) to (6) is 3 + 5 = 8 minutes

Travel time of 8 minutes from (8) to (6) is less than previous travel time of
9999 minutes at (6), updating (6) predecessor to (8) and min travel time to 8
minutes

Added (6) to queue: Front <- (7) | (4) | (6) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df513d0>


Checking neighbor: (7)

Travel time from (8) to (7) updated to 5 minutes

Travel time from (7) to (8) updated to 5 minutes

Total travel time from (8) to (7) is 3 + 5 = 8 minutes

Travel time of 8 minutes from (8) to (7) is NOT less than previous travel time
of 4 minutes at (7), do nothing
```

```
(7) has already been visited or is already in the queue: Front <- (7) | (4) |
(6) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dabd410>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (7) | (4) | (6) <- Rear

Visiting node: (7), popped the node from queue: Front <- (4) | (6) <- Rear

Getting neighbors: [(1)_Movement_Boulder, (8)_Whole_Foods_Market,
(6)_Peleton_West]

Checking neighbor: (1)

Total travel time from (7) to (1) is 4 + 4 = 8 minutes

Travel time of 8 minutes from (7) to (1) is NOT less than previous travel time
of 0 minutes at (1), do nothing

(1) has already been visited or is already in the queue: Front <- (4) | (6) <-
Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df94850>


Checking neighbor: (8)

Total travel time from (7) to (8) is 4 + 5 = 9 minutes

Travel time of 9 minutes from (7) to (8) is NOT less than previous travel time
of 3 minutes at (8), do nothing

(8) has already been visited or is already in the queue: Front <- (4) | (6) <-
Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df64310>


Checking neighbor: (6)

Travel time from (7) to (6) updated to 5.5 minutes

Travel time from (6) to (7) updated to 5.5 minutes
```

```
Total travel time from (7) to (6) is 4 + 5.5 = 9.5 minutes

Travel time of 9.5 minutes from (7) to (6) is NOT less than previous travel time
of 8 minutes at (6), do nothing

(6) has already been visited or is already in the queue: Front <- (4) | (6) <-
Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dabe610>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (4) | (6) <- Rear

Visiting node: (4), popped the node from queue: Front <- (6) <- Rear

Getting neighbors: [(1)_Movement_Boulder, (2)_Mount_Sanitas_Trailhead,
(3)_Chautauqua_Park, (5)_Fiske_Planetarium, (8)_Whole_Foods_Market]

Checking neighbor: (1)

Total travel time from (4) to (1) is 7 + 7 = 14 minutes

Travel time of 14 minutes from (4) to (1) is NOT less than previous travel time
of 0 minutes at (1), do nothing

(1) has already been visited or is already in the queue: Front <- (6) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dac15d0>


Checking neighbor: (2)

Travel time from (4) to (2) updated to 4 minutes

Travel time from (2) to (4) updated to 4 minutes

Total travel time from (4) to (2) is 7 + 4 = 11 minutes

Travel time of 11 minutes from (4) to (2) is less than previous travel time of
9999 minutes at (2), updating (2) predecessor to (4) and min travel time to 11
minutes

Added (2) to queue: Front <- (6) | (2) <- Rear
```

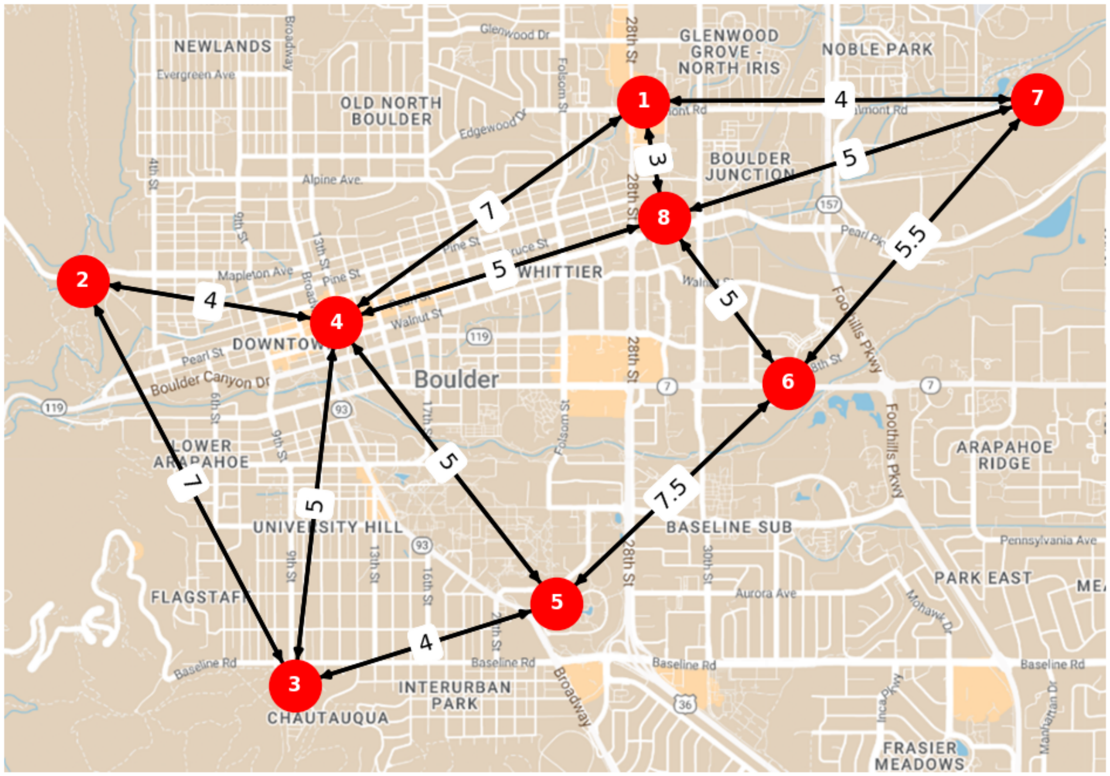
```
Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93e0b4950>


Checking neighbor: (3)

Travel time from (4) to (3) updated to 5 minutes

Travel time from (3) to (4) updated to 5 minutes

Total travel time from (4) to (3) is 7 + 5 = 12 minutes

Travel time of 12 minutes from (4) to (3) is less than previous travel time of
9999 minutes at (3), updating (3) predecessor to (4) and min travel time to 12
minutes

Added (3) to queue: Front <- (6) | (2) | (3) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df65610>


Checking neighbor: (5)

Travel time from (4) to (5) updated to 5 minutes

Travel time from (5) to (4) updated to 5 minutes

Total travel time from (4) to (5) is 7 + 5 = 12 minutes

Travel time of 12 minutes from (4) to (5) is less than previous travel time of
9999 minutes at (5), updating (5) predecessor to (4) and min travel time to 12
minutes

Added (5) to queue: Front <- (6) | (2) | (3) | (5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93e0d5550>


Checking neighbor: (8)

Total travel time from (4) to (8) is 7 + 5 = 12 minutes

Travel time of 12 minutes from (4) to (8) is NOT less than previous travel time
of 3 minutes at (8), do nothing

(8) has already been visited or is already in the queue: Front <- (6) | (2) |
(3) | (5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dac0250>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (6) | (2) | (3) | (5) <- Rear

Visiting node: (6), popped the node from queue: Front <- (2) | (3) | (5) <- Rear

Getting neighbors: [(5)_Fiske_Planetarium, (8)_Whole_Foods_Market,
(7)_Valmont_Dog_Park]

Checking neighbor: (5)

Travel time from (6) to (5) updated to 7.5 minutes

Travel time from (5) to (6) updated to 7.5 minutes

Total travel time from (6) to (5) is 8 + 7.5 = 15.5 minutes

Travel time of 15.5 minutes from (6) to (5) is NOT less than previous travel
time of 12 minutes at (5), do nothing

(5) has already been visited or is already in the queue: Front <- (2) | (3) |
(5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dfc7810>


Checking neighbor: (8)

Total travel time from (6) to (8) is 8 + 5 = 13 minutes

Travel time of 13 minutes from (6) to (8) is NOT less than previous travel time
of 3 minutes at (8), do nothing

(8) has already been visited or is already in the queue: Front <- (2) | (3) |
(5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dfcb510>


Checking neighbor: (7)
```

```
Total travel time from (6) to (7) is 8 + 5.5 = 13.5 minutes

Travel time of 13.5 minutes from (6) to (7) is NOT less than previous travel
time of 4 minutes at (7), do nothing

(7) has already been visited or is already in the queue: Front <- (2) | (3) |
(5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dfb3510>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (2) | (3) | (5) <- Rear

Visiting node: (2), popped the node from queue: Front <- (3) | (5) <- Rear

Getting neighbors: [(3)_Chautauqua_Park, (4)_Pearl_Street_Mall]

Checking neighbor: (3)

Travel time from (2) to (3) updated to 7 minutes

Travel time from (3) to (2) updated to 7 minutes

Total travel time from (2) to (3) is 11 + 7 = 18 minutes

Travel time of 18 minutes from (2) to (3) is NOT less than previous travel time
of 12 minutes at (3), do nothing

(3) has already been visited or is already in the queue: Front <- (3) | (5) <-
Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dac1e50>


Checking neighbor: (4)

Total travel time from (2) to (4) is 11 + 4 = 15 minutes

Travel time of 15 minutes from (2) to (4) is NOT less than previous travel time
of 7 minutes at (4), do nothing

(4) has already been visited or is already in the queue: Front <- (3) | (5) <-
Rear

Current Status of Graph (table format):
```

```
<pandas.io.formats.style.Styler at 0x2d93e0b5610>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (3) | (5) <- Rear

Visiting node: (3), popped the node from queue: Front <- (5) <- Rear

Getting neighbors: [(2)_Mount_Sanitas_Trailhead, (4)_Pearl_Street_Mall,
(5)_Fiske_Planetarium]

Checking neighbor: (2)

Total travel time from (3) to (2) is 12 + 7 = 19 minutes

Travel time of 19 minutes from (3) to (2) is NOT less than previous travel time
of 11 minutes at (2), do nothing

(2) has already been visited or is already in the queue: Front <- (5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93dfb1b10>


Checking neighbor: (4)

Total travel time from (3) to (4) is 12 + 5 = 17 minutes

Travel time of 17 minutes from (3) to (4) is NOT less than previous travel time
of 7 minutes at (4), do nothing

(4) has already been visited or is already in the queue: Front <- (5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93e099710>


Checking neighbor: (5)

Travel time from (3) to (5) updated to 4 minutes

Travel time from (5) to (3) updated to 4 minutes

Total travel time from (3) to (5) is 12 + 4 = 16 minutes

Travel time of 16 minutes from (3) to (5) is NOT less than previous travel time
of 12 minutes at (5), do nothing
```

```
(5) has already been visited or is already in the queue: Front <- (5) <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93e021190>


Sorting queue by time attribute of node (must visit next closest node first):
Front <- (5) <- Rear

Visiting node: (5), popped the node from queue: Front <-  <- Rear

Getting neighbors: [(3)_Chautauqua_Park, (4)_Pearl_Street_Mall,
(6)_Peleton_West]

Checking neighbor: (3)

Total travel time from (5) to (3) is 12 + 4 = 16 minutes

Travel time of 16 minutes from (5) to (3) is NOT less than previous travel time
of 12 minutes at (3), do nothing

(3) has already been visited or is already in the queue: Front <-  <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df53c90>


Checking neighbor: (4)

Total travel time from (5) to (4) is 12 + 5 = 17 minutes

Travel time of 17 minutes from (5) to (4) is NOT less than previous travel time
of 7 minutes at (4), do nothing

(4) has already been visited or is already in the queue: Front <-  <- Rear

Current Status of Graph (table format):

<pandas.io.formats.style.Styler at 0x2d93df51310>


Checking neighbor: (6)

Total travel time from (5) to (6) is 12 + 7.5 = 19.5 minutes

Travel time of 19.5 minutes from (5) to (6) is NOT less than previous travel
time of 8 minutes at (6), do nothing

(6) has already been visited or is already in the queue: Front <-  <- Rear
```

```
-----------------------------
---- Algorithm Complete! ----
-----------------------------

The minimum travel time from (1)_Movement_Boulder to (3)_Chautauqua_Park is 12
minutes, following the route ['(1)', '(4)', '(3)'].

<pandas.io.formats.style.Styler at 0x2d93e0157d0>
```



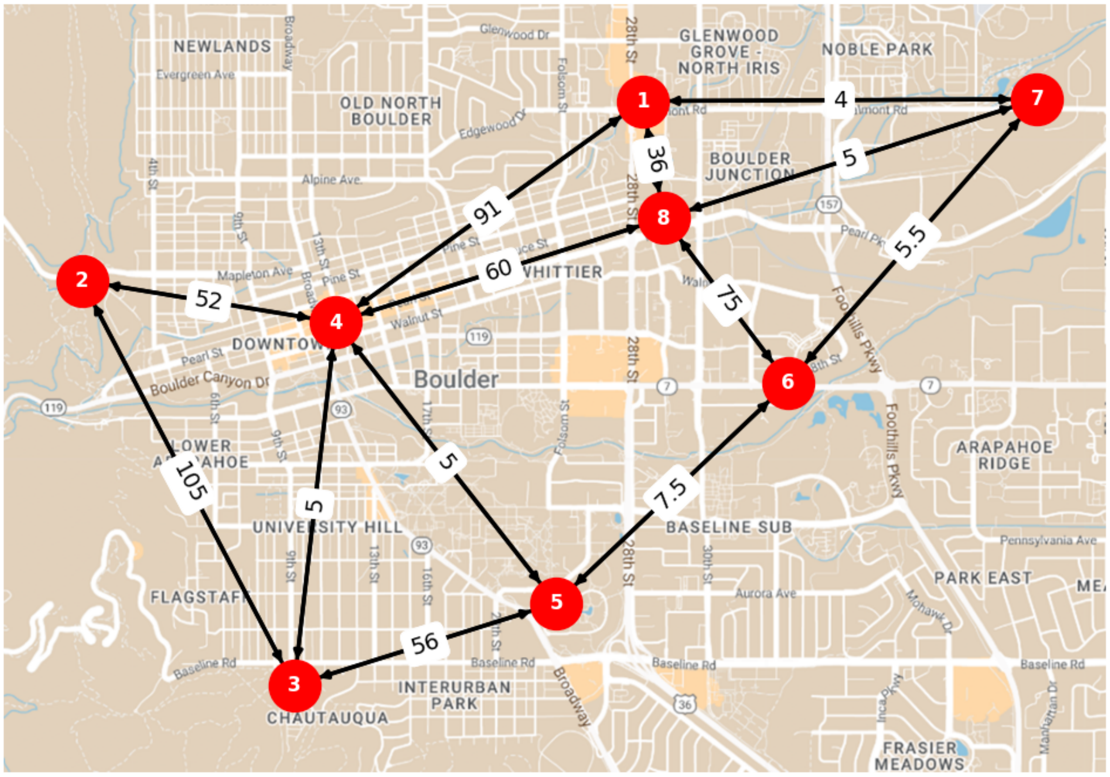
Baseline + Uncertainty Results Discussion

The code for testing our baseline case plus uncertainty (no rush hour scaling) is shown below along with the corresponding results. For this case, the model had quite a few routes that ended up being affected by the uncertainty (representing unexpected road work, accidents, etc.) - showing that our code to incorporate uncertainty is working. See routes 1 to 4, 1 to 8, 4 to 8, 8 to 6, 2 to 4, 2 to 3, and 3 to 5 (lots of accidents happened this day... could be the start of the apocolypse). All have been modified with a scaling factor between 10-15. This drastically changed the optimal route to take, changing from the base of 1, 4, 3 to 1,7,6,5,4,3 to avoid all the routes that had been affected. This is clearly due to accidents between nodes 1 and 4 as well as 1 and 8, making a roundabout path through node 7 a better option. Additional accidents between 3 and 5 and 4 and 8 also pushed the route down through 6 then back up through 4. This resulted in a total time of 27 minutes to get to the same location that took just 12 minutes in the base case. Again, we did manual calculations to verify this is the most efficient route to take given our graph.

```
[ ]: # Reset all the nodes back to initial states
     for node in node_list:
         node.reset_node()

     # Define the starting node, set travel time to 0 (already here), the end node,␣
      ↪and if is rush hour
     start_node            = node_01
     start_node.set_time(0)
     end_node              = node_03
     is_rush_hour          = False
     include_uncertainty = True

     # Find the minimum travel time and path
     min_time, path = find_min_travel_time(start_node, end_node, is_rush_hour,␣
      ↪include_uncertainty, node_list)

     # Print the results
     print(f'The minimum travel time from {start_node.get_name_with_id()} to␣
      ↪{end_node.get_name_with_id()} is {min_time} minutes, following the route␣
      ↪{path}.')

     # Print the final graph data as a table
     print_table(start_node, node_list)

     # Visualize the graph
     visualize_graph(node_list)
```

```
The minimum travel time from (1)_Movement_Boulder to (3)_Chautauqua_Park is 27.0
minutes, following the route ['(1)', '(7)', '(6)', '(5)', '(4)', '(3)'].
```

```
<pandas.io.formats.style.Styler at 0x2d93c170450>
```



Rush Hour + Uncertainty Results Discussion

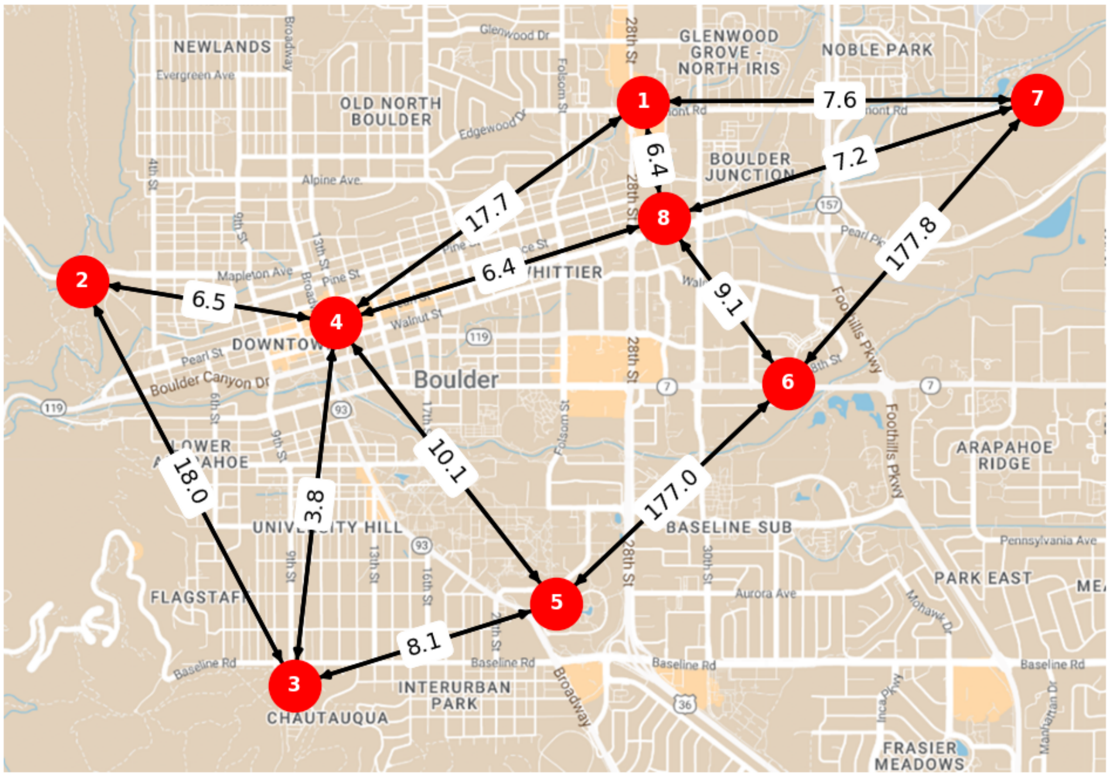
The code for testing our rush hour case plus uncertainty is shown below along with the corresponding results. For this scenario, our route was affected by both rush hour and uncertainty (accidents, road work, etc.). You can see both are working as intended in this scenario as the rush hour routes are scaled by a multiplier dependent on the standard normal distribution with a mean of 2 and standard deviation of .5 (see helper function above that shows this) and the uncertainty is once again a random multiplier between 10-15. Connections that are affected by the uncertainty have greater delays than those of the rush hour multiplier (because they have a higher scaling factor). Note that it is possible for a route to be affected by both rush hour and uncertainty scaling (see from 5 to 6 or 6 to 7 for examples); in those scenarios the final scaling factor is the product of rush hour and uncertainty scaling factors. In this scenario, our optimal route changed again from the base case and the optimal route found by the algorithm is 1,8,4,3 with a total time of 16.6 minutes. We verified this to be true with manual calculations once again. This route changed because traffic was bad between 1 and 4, making 1 to 8 to 4 a shorter route.

```
[ ]: # Reset all the nodes back to initial states
     for node in node_list:
         node.reset_node()

     # Define the starting node, set travel time to 0 (already here), the end node,␣
      ↪and if is rush hour
     start_node             = node_01
```

```
start_node.set_time(0)
end_node              = node_03
is_rush_hour          = True
include_uncertainty = True

# Find the minimum travel time and path
min_time, path = find_min_travel_time(start_node, end_node, is_rush_hour,␣
  ↪include_uncertainty, node_list)

# Print the results
print(f'The minimum travel time from {start_node.get_name_with_id()} to␣
  ↪{end_node.get_name_with_id()} is {min_time} minutes, following the route␣
  ↪{path}.')

# Print the final graph data as a table
print_table(start_node, node_list)

# Visualize the graph
visualize_graph(node_list)
```

```
The minimum travel time from (1)_Movement_Boulder to (3)_Chautauqua_Park is 16.6
minutes, following the route ['(1)', '(8)', '(4)', '(3)'].
```

```
<pandas.io.formats.style.Styler at 0x2d941d36110>
```


1
2
3
4
5
6
7
8
7.6
7.2
6.4
17.7
6.4
6.5
9.1
177.8
18.0
3.8
10.1
177.0
8.1
NEWLANDS
OLD NORTH BOULDER
GLENWOOD GROVE - NORTH IRIS
NOBLE PARK
BOULDER JUNCTION
WHITTIER
DOWNTOWN
Boulder
UNIVERSITY HILL
FLAGSTAFF
BASELINE SUB
ARAPAHOE RIDGE
PARK EAST
INTERURBAN PARK
CHAUTAUQUA
FRASIER MEADOWS