Project Proposal
CS 5800
December 12, 2022

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Application of Graph Algorithms for Running Route Optimization

Table of Contents:

Abstract

Within the domain of computer science, many classical algorithms especially within the confines of graph traversal have had a major impact on everyday life when it comes to optimization to save time and enhance experiences. In recent years, many of these algorithms have been developed into major platforms that focus on the optimization of routes to enhance driving experience (Google Maps, Apple Maps, Waze), however, very few focus on runners, especially those in dense cities. To address this need, we have prepared a proof of concept (POC) application that utilizes 5 different algorithms designed to enhance the user experience of runners using 5 separate use cases.

Project Proposal
CS 5800
December 12, 2022

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Introduction

One of the most famous events that takes place in the city of Boston is the annual Boston Marathon that attracts runners from around the world to participate in a challenging 26-mile jog. Although professional runners from across the globe participate in this event, many locals in the Boston area do as well. Similarly to many other dense and vibrant cities such as Boston, one of the biggest challenges is finding appropriate training routes to practice within. The purpose of this project, and our main goal, is to utilize many of the algorithms we covered within the confines of this course to help optimize the running experience. Within this project, we will evaluate the use of several algorithms, in a proof-of-concept (POC) fashion, to address a number of shortcomings that runners in dense cities tend to experience.

Many of these short-comings can be grouped into three main categories: (1) finding an interesting route to run long distances within a smaller dense neighborhood, (2) navigating a dense city by finding the shortest path to a destination you want to run to, (3) being able to optimize a route to visit certain interesting sites along your run. In order to address these, we decided to implement a number of algorithms covered in this course, but altered slightly to suit our use cases here.

Personal Significance – Saleh

One of the most famous events that takes place in the city of Boston is the annual Boston Marathon. Although runners from around the globe participate in the event, many locals in the Boston area do as well. One of the biggest challenges for runners in vibrant cities such as Boston is finding appropriate training routes to practice for the marathon. The purpose of this project is to develop a proof-of-concept model utilizing various algorithms to help optimize a runner's training path. This project is personally relevant to me for two main reasons: First, I am a runner, and live in Boston, and have always found it challenging to find new and interesting routes to keep my run both motivating and optimal in length. The second motivating factor on my end is that I focus many aspects of my life on optimization. I run a medium-sized data science team, and we spend a considerable amount of time optimizing machine learning models, but one area I am not as well versed in is optimizing graph networks. That said, I would like to use this opportunity to explore a new and novel area for me when it comes to graphs.

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Personal Significance – Gabriel

Health issues, especially those around eating disorders like being overweight, can develop into multiple and very impactful diseases such as diabetes and coronary issues. One of the main ways to combat this rising epidemic, is exercising and cardiovascular exercise, has a very beneficial effect on overall health and particularly the increased release of endorphins, helps with taking back control on what we eat. In my family there's a history of heart disease, so managing my food intake and exercise routine is crucial for me. One of the best cardiovascular exercises that works for me is running in a beautiful environment where I can just relax and enjoy the scenery. Since I'm now moving to Boston, it would be of great help to find training routes that would optimize the "beautifulness" given the distance that I intend to run.

Personal Significance – Nile

Distance training is already a difficult task, trying to find space to train does not need to be part of the problem. There exists equipment on the market like treadmills and indoor bikes that allow the user to exercise indoors. The problem with those is the cost and also space. Living in the city there is a tradeoff of less space for convenience. A solution to this problem is to exercise outside. A method to training is keeping track of progress, this is done by recording the amount of time to complete a certain amount of distance. From this project I hope to explore finding the best route for a certain distance and avoid areas of construction or other factors that would make training less ideal. This algorithm would also benefit people who are traveling and are not familiar with the area to keep up with their training. I look forward to finding an efficient solution to this problem.

Analysis

When it comes to the analysis, given that five different algorithms and subsequent models were implemented to address different use cases and user needs, we have broken down this section so that each can be addressed comprehensively and independently. For each of the methods and algorithms, we introduced the problem statement, the method, the rationale, and an explanation of the algorithm and its associated time complexity. With each of these cases, we show its application using data that was prepared and gathered, which will be the first topic of discussion within this section.

Data Gathering and Preparation

When it comes to data gathering, we explored a number of methods to draw and generate maps that can be used to demonstrate the utility and application of our algorithms for the specific use cases at hand. In an ideal world with enough time and resources, a graph database such as MarkLogic, Neo4j, or

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Stardog would have been useful to store the content of our data representing streets and intersections. However, given the scope and timeline of this project, we used a standard CSV file to store our data using four columns: the start node, destination node, distance, and importance which represents the popularity of that site. With the schema prepared, we undertook the task of populating it with data. Using GoogleEarth, we isolated a small neighborhood near Northeastern's campus, and populated the CSV with the information we gathered. You can see a sample of the data shown in the Table below:

source	destination	distance	importance
0	1	11	1
1	2	12	1
2	3	11	2

Table 1: A table representing an example of how the data was stored

We then used a Python library called networkx to log and draw a graphical representation of the map. We can see an example of this map in the figure below. To preprocess our data, we used Jupyter Notebooks as our main tool which is a web-based interactive notebook specific for the development of models and algorithms. First, we imported our libraries as shown in the figure below:

```
import networkx as nx
import matplotlib.pyplot as plt
import sys
import pandas as pd
from collections import defaultdict, deque
```

Figure 1: Example libraries being imported into the notebook

Next, the data is imported and read in using a Pandas dataframe, a data structure commonly used in the data science domain. We can see an example of this below:

```
df = pd.read_csv("../../data/boston_traced_map_norm.csv")
df[:5]
```

Figure 2: Example of the data being imported into the notebook

Finally, we then create a Graph object using the variable G, and iterate over the dataframe and subsequently populate the graph with our data of interest. We can see the steps for this process below:

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

```
G = nx.Graph()

for row in df.values:
    G.add_edge(row[0],row[1], weight=row[2])

nx.draw(G, with_labels=True)
plt.show()
```

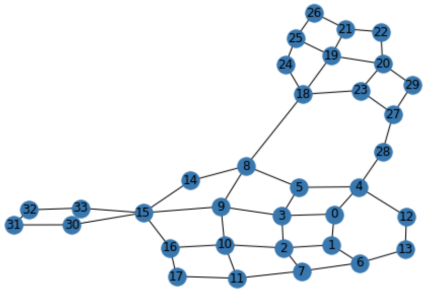


Figure 3: Example of the graph being constructed and drawn (Not to scale)

With the graph constructed, we were now able to use this as a starting point for the application of some of our algorithms. It is important to note that not all of the algorithms presented here used this data structure as a starting point, and therefore we exported this data in various formats such as adjacency lists, adjacency matrices, edge lists, and several others.

Method 1: Identification of a Longest Limited Path for Long-Distance Runners

The longest path problem is a well known area of research that involves the identification of a path of maximum length within a given network of nodes and edges. The application of a true longest path problem is NP-Hard, as opposed to its shortest path counterpart which can be solved in Polynomial time. Our objective with regards to this algorithm is to help a runner find a long path, in a small dense network, limited by a user's preferred distance in order to give them an ideal running path to train on.

We implemented an algorithm known as Breadth-first search (BFS) to help with this issue. The main idea with BFS is that unlike with trees, graphs can contain cycles in which the traversal of a node may occur more than once. The idea here is to avoid this by noting which nodes were visited, and with this in mind, we can identify a path of interest. However, the main objective here is not necessarily limiting the path, but finding a long route limited by the users input represented as either the number of nodes or traffic intersections they want to cross, or limited by the total distance that they will run. Therefore, we developed the following algorithm to address this issue:

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

```python
def boston_longest_limited_distance(graph, end_node, limit):
    """
    Function that uses Breadth-First Search (BFS) to find the longest limited route
    @input graph: The graph of the network
    @input end_node: Final node to end at
    @input limit: Maximum distance

    Returns the total distance as well as the path to take
    """
    # Create a dictionary
    def_d = defaultdict(list)
    # Create a queue using deque
    queue = deque([('0', 0, [])])

    # Create the result variable to store the results
    result = {end_node:(0, [])}

    # Start for loop to iterate over the graph, store nodes and values
    for start, *args in graph:
        def_d[start].append(args)

    # Enter while loop based on queue
    while queue:
        start, args, c = queue.popleft()

        # If limit is reached
        if start == end_node and len(c) == limit:
            if result[start][0] < args:
                result[start] = (args, c)

        #  If limit is NOT yet reached
        if len(c+[start]) <= limit:
            for x, y in def_d[start]:
                queue.append((x, args+y, c+[start]))

    print(result[end_node])
```

Figure 4: Method 1 Algorithm for the Longest Limited Path

The idea here is to create a dictionary and queue to organize the nodes. We then iterate over the graph, store the values, and then enter a second set of loops that store the distance so far. If the distance limit is reached, we stop, otherwise continue. Given the nature of the algorithm, and its dependence on V and E representing the nodes and edges of this graph, we can determine that the total runtime will be O(|V| + |E|) since the network's size is the main driver here. Given our use case of focusing on smaller neighborhoods in the Boston area, within running distance, we do not anticipate users reaching values high enough when it comes to nodes and edges. Upon implementing this algorithm using our dataset, we arrived at the following results:

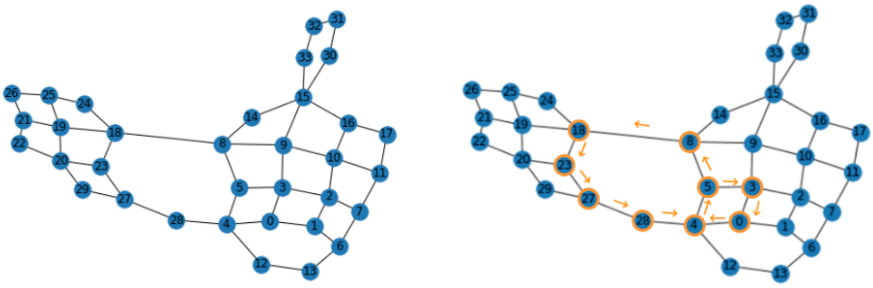


Figure 5: Graph before and after the application of the algorithm (Not to scale)

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

In summary, we managed to successfully develop an algorithm in Python that allows the user to find a long path in a given neighborhood which can be limited in distance based on the user's preferences. We demonstrated this with two examples in our code showing the algorithm taking in the users input, and returning the path of interest.

Method 2: Application of Dijkstra's Algorithm for Short Distance Runners

The shortest path problem has the objective of finding a path between two nodes in a given graph network in which the sum of the weights between the source and destination is minimized relative to other alternatives. There are a number of different algorithms that have been developed and explored to solve this problem such as Bellman-Form, Floyd-Warshall, and Dijkstra's, the third of which will be the focus of this section.

In the previous method, we supported our long-distance runners by finding longer paths. In this method, we will support our short distance runners by finding the shortest path based on distance between two nodes. To accomplish this, we created a Python script to implement Dijkstra's Algorithm.

The main idea behind this algorithm is to find the shortest path between two given nodes in a graph. There are many variants to this algorithm but the main concept is to have a source node, which iteratively traverses neighboring nodes until a path to the destination node is reached which minimizes the distance. We can accomplish this in a few steps: first we keep track of our visited nodes by marking them all as unvisited at the beginning of the algorithm. Next set the initial distance from the source to target as infinite, and then calculate the distances of neighbor nodes of the active node. Next the sum distance with weights is calculated from the path. If the calculated distance is smaller, we update the distance and set the current node, and keep repeating until no nodes are left.

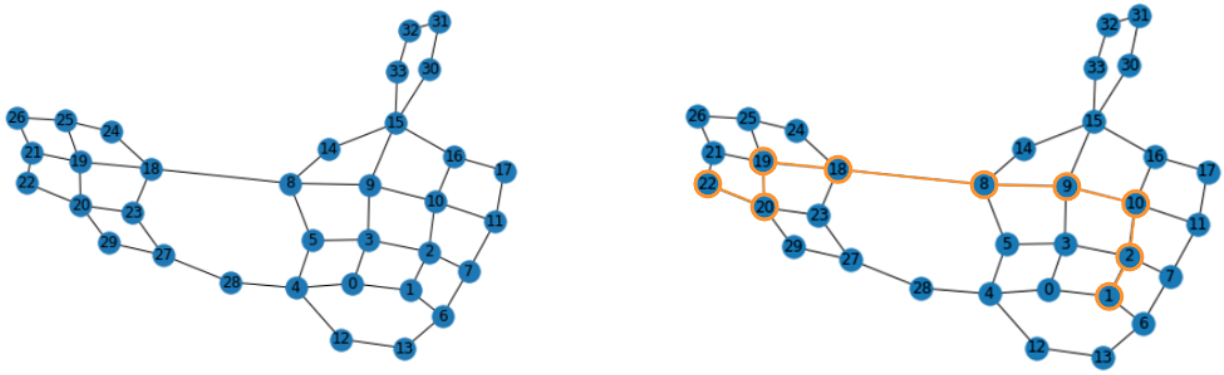


Figure 6: Graph before and after the application of the algorithm (Not to scale)

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Within this algorithm, the time complexity as the result of reach edge is O(|E|), and the complexity of each vertex is O(|V|). Given our implementation which uses the output values of our networkx object, we reached a time complexity of O(|E| + |V|). A future improvement could be the use of a priority queue in order to reach a time complexity of O(|E| + |V| log |V|).

In summary, we successfully developed an algorithm to support our group of users that are short distance runners. We used Dijkstra's algorithm to find the shortest path between two nodes in a given graph network. We demonstrated the utility of our algorithm in our Python code showing two examples in which the correct path was successfully identified.

Method 3: Visiting Specific Sites with Minimal Distance

As a runner, I often find myself wanting to establish a routine route in which I visit certain sites that I can use as landmarks within the city of Boston. This not only helps me maintain an adequate pace by not having to stop and get directions, but also enhances my experience by taking advantage of the wonderful sites the city has to offer. For this particular use case, our objective was to implement a greedy algorithm from networkx as an application for the Travelling Salesman Problem within our graph.

The algorithm operates as follows: First, we instantiate a few variables to maintain the data, specifically a list that will hold the indices of the nodes of interest, as well as a space to store the results containing the nodes. Next we traverse the adjacency matrix used to store the data, and for every node if the cost to reach it is less than the current one, we update the current cost. Finally, we generate a minimum path and return it to the user. Given the methodology of this algorithm, we can expect the time complexity to be O(n^2 log(n)). We can see a working example of this in the figure below. Please note that this algorithm was taken directly from the networkx library using the approximation class.

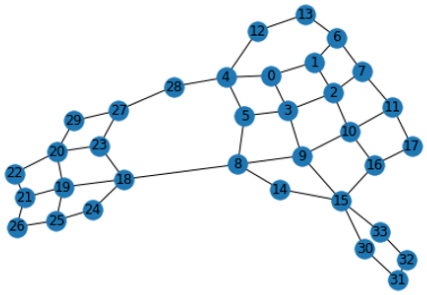
Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Create Graph:

```
In [11]: G = nx.Graph()

In [12]: for row in df.values:
             G.add_edge(row[0],row[1], weight=row[2])

In [15]: nx.draw(G, with_labels=True)
         plt.show()
```



Apply TSP Algorithm:

```
In [16]: tsp = nx.approximation.traveling_salesman_problem

In [21]: tsp(G, nodes=[0, 20, 18, 9, 27])
Out[21]: [0, 4, 28, 27, 23, 20, 19, 18, 8, 9, 3, 0]
```

Figure 7: Graph showing the application of the algorithm (Not to scale)

In this example, we specify that we want to visit nodes 0, 20, 18, 9, and 27. We can see the output of the ideal path shown in the diagram as well. We can graphically visualize the results using the figure below:

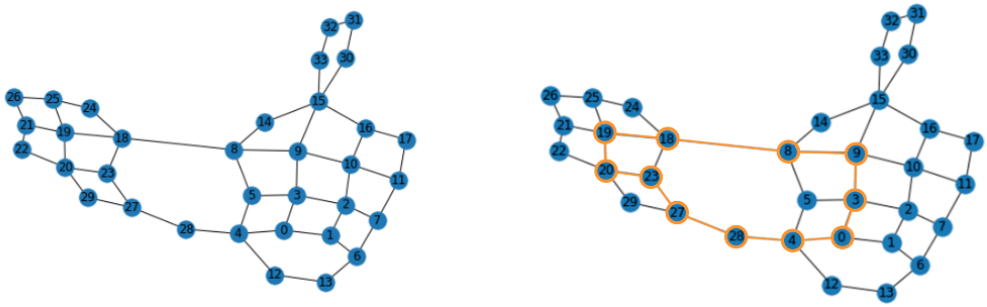


Figure 8: Graph before and after the application of the algorithm (Not to scale)

The traveling salesman problem is a well known and well studied problem in the first of algorithms and computer science. In this project, our objective was to implement a greedy approach to solve this issue as a means of providing runners with the ability to target certain sites in a shortest path fashion to enhance the quality and experience of their run.

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

In summary, we successfully implemented a greedy approach to the traveling salesman problem to enable users to visit certain sites along their run. The algorithm allows users to enter the nodes of the specific sites they want to visit, and creates an ideal path or route for them to use.

Method 4: Highest Quality Roads for a Given Distance

When training and tracking progress people like to run for a set distance and monitor how long it takes to complete the distance. Also, when given an option of two paths people would like to choose the "better" option. Living in an area that experiences severe weather and the sidewalk conditions vary from location.

The idea of this method is to find the best options of path for a set amount of distance. Finding the best path can be done by applying another weight to the road which would represent the quality coefficient and the quality of the road is the product of $Q_R = Q_{CO} * R_D$. Where Q_{CO} is the quality coefficient and R_D is the road distance. To find the overall quality of the path is the result of $Q_P = \Sigma Q_R / \Sigma R_D$. The quality coefficient is a value ranging from 0 to 1, where values closer to 1 indicate "good" conditions. This method takes the sum of the quality of roads in the path and divides it by the sum of the total distance of the path. The total distance of the path is also the maximum score of the sum of the quality of roads. This is done to cover situations where the set distance is a range. If two possible paths both have perfect conditions but one of longer by .1 the longer distance path would have a higher sum of quality of roads. By dividing by the total distance of the road, it can give a more accurate result.

A requirement for this method was for a path to be a possible solution it would have to produce a cycle. Where the first node needs to equal the last node in the traveled list. Another requirement was that when searching for the next adjacent node to add, it can not be the same as the previous node. This will prevent solutions of running back and forth on the same road multiple times.

The algorithm for the method uses similar methods as depth first search inorder to find possible paths. It will keep expanding the path list until it surpases the distance limit. When it goes over the distance limit it will remove the previous node and try another option. To save time and not have to recalculate the distance and quality, the algorithm implements memorization. If a path is found to produce the cycle explained earlier and is in the distance range then the path is compared to the current set max quality path. If the new path is larger then the new path takes the place as the new max.
By implementing memorization the algorithm's time complexity is O(N + E). where N is the number of nodes and E is the number of edges. Overall this is a great algorithm to find the best route for a set distance.

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Method 5: Visiting Interesting sites in a given distance.

One of the most motivating parts of any activity of long duration is reaching small goals. To make the run as interesting as possible, we have implemented an algorithm that given a distance, it will return a route that visits the maximum number of interesting sites. This way, the training becomes more fun and mentally manageable.

We could have used straight up BFS, but in a very connected graph, the complexity and space could easily become unmanageable if the distance of the run is a couple of miles long.
In order to address this situation, we first reduced the graph into a complete graph containing only the starting point and interesting sites as nodes and the edges are a representation of the shortest path between all nodes. This shortest path was created using Djikstra's algorithm. One important aspect to mention is that when looking for the shortest path, we did not remove the already used edges on previous shortest paths, so we are actually allowing for edge reutilization. Let's say that the shortest path from node 1 to node 2 uses the edges a, b and c and the shortest path between node 1 and 3 is a, b, d, then on the reduced graph, the edge connecting node 1 to node 2 is a representation of the path a, b and c and the edge connecting the node 1 with the node 3 is a representation of the edges a, b and d.

Since this reduced graph is a complete graph where all nodes are interesting sites or the source and all edges are the shortest paths of the original graph, then using a greedy approach of choosing the closest node iteratively, will give us the maximum number of interesting sites where the total distance is less than or equal to the one provided by the user.
We did apply the constraint of not repeating an interesting site on the greedy algorithm.
The first image represents the original graph where the green node is the starting point, the red nodes are the interesting sites and the blue nodes are not interesting nodes, but part of the map.
The second image is the reduced graph and the greedy algorithm's route in green and finally, the third graph is the reconstruction of the optimized route into the original graph.

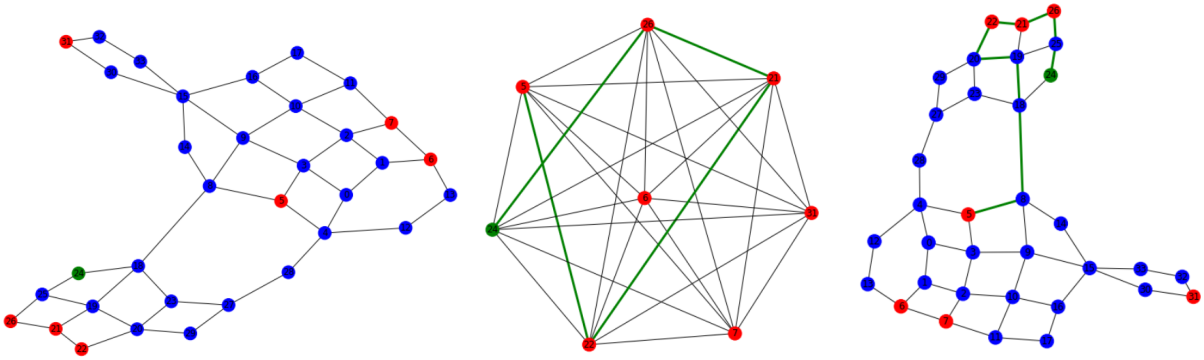


Figure 9: Graph before and after the application of the algorithm (Not to scale)

Djikstra's algorithm is applied between all nodes in the reduced graph, so in the worse case scenario, if all nodes are interesting nodes, then we have V^2 applications of Dijkstra which itself has a complexity of

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

$O(E + VlogV)$ and the greedy algorithm has a worse case scenario of the reduced edges squared so V'^2 where V' is the number of interesting sites and this will be V^4 of the original graph nodes. This give a worse case complexity of $O(V^3 log(V) + V^2 \times E + V^4) = O(V^4)$. But in an average case, the interesting sites will be << than the number of nodes, so the reduction of the graph has a complexity of $O(V' \times V \times log(V) + V' \times E + V'^2)$.

Conclusion

Within this proof of concept, our objective was to answer the question of whether or not a number of graph algorithms we learned over the course of this semester can be used to address the many unmet needs for runners in cities such as Boston. Over the course of this project, we developed and implemented 5 different algorithms that supported 5 different use cases for runners. Together, we will now summarize each of the methods we implemented with respect to the end user: Method 1 enables long distance runners to find a long path in a small neighborhood and limit the distance based on preference. Method 2 allows short distance runners to find the shortest path possible to a destination of interest. Method 3 gives the user the ability to visit interesting sites within the shortest distance. Method 4 allows users to find routes with the highest quality sidewalks or paths with the least traffic to enhance their run. Method 5 enables users to generate an ideal run path to visit interesting sites in the city of Boston to keep them motivated. While our methods may be linked, they are very distinct in functionality. Not only did we demonstrate an example for each of the algorithms, but also extracted a neighborhood near Northeastern's campus to use in our feasibility assessments. We implemented each of the algorithms using this dataset to not only demonstrate how the algorithms work, but also how they can be used in a real-life scenario in the city of Boston. With this, we were able to successfully answer our question in a positive manner, with real-life data, and show how the many algorithms we covered within this course can be used to support runners both in the city of Boston and around the world.

Appendix

Important Links

YouTube Video

https://youtu.be/_IN50KNwwzs
https://www.youtube.com/watch?v=H-1O2Uq4sDg&feature=youtu.be

GitHub Repository

https://github.com/alkhalifas/boston-route-optimization

Presentation Slides

https://github.com/alkhalifas/boston-route-optimization/blob/main/Project%20Presentation.pptx

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Code - Method 1

Import Libraries:

```
import networkx as nx
import matplotlib.pyplot as plt
import sys
import pandas as pd
from collections import defaultdict, deque
```

Import Data:

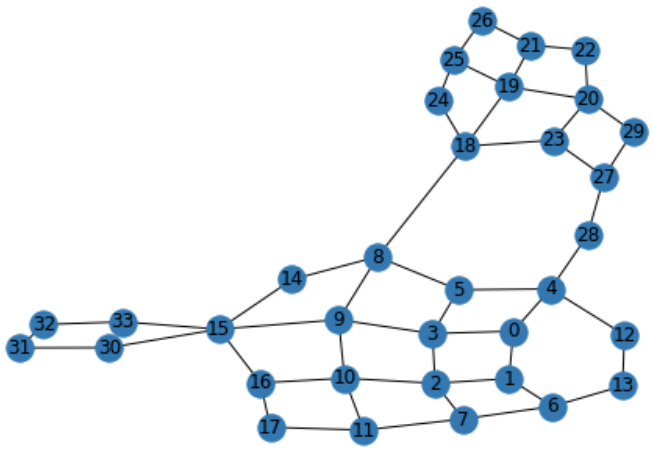
```
df = pd.read_csv("../../data/boston_traced_map_norm.csv")
df[:5]
```

	s	d	w
0	0	1	11
1	1	2	12
2	2	3	11
3	0	4	14
4	3	0	10

```
G = nx.Graph()
```

```
for row in df.values:
    G.add_edge(row[0],row[1], weight=row[2])
```

```
nx.draw(G, with_labels=True)
plt.show()
```



Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

```
def boston_longest_limited_distance(graph, end_node, limit):
    """
    Function that uses Breadth-First Search (BFS) to find the longest limited route
    @input graph: The graph of the network
    @input end_node: Final node to end at
    @input limit: Maximum distance

    Returns the total distance as well as the path to take
    """
    # Create a dictionary
    def_d = defaultdict(list)

    # Create a queue using deque
    queue = deque([('0', 0, [])])

    # Create the result variable to store the results
    result = {end_node:(0, [])}

    # Start for loop to iterate over the graph, store nodes and values
    for start, *args in graph:
        def_d[start].append(args)

    # Enter while loop based on queue
    while queue:
        start, args, c = queue.popleft()

        # If limit is reached
        if start == end_node and len(c) == limit:
            if result[start][0] < args:
                result[start] = (args, c)

        #  If limit is NOT yet reached
        if len(c+[start]) <= limit:
            for x, y in def_d[start]:
                queue.append((x, args+y, c+[start]))

    print(result[end_node])

boston_longest_limited_distance(Graph_Boston, '13', 22)
```

```
(257.0, ['0', '4', '5', '3', '0', '4', '5', '3', '0', '4', '5', '3', '0', '4', '5', '8',
'18', '23', '27', '28', '4', '12'])
```

```
boston_longest_limited_distance(Graph_Boston, '1', 12)
```

```
(140.0, ['0', '4', '5', '8', '18', '23', '27', '28', '4', '5', '3', '0'])
```

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Code - Method 2

Import Libraries:

```
from queue import PriorityQueue
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import sys
import pandas as pd
from collections import defaultdict, deque
```

Import Data

```
df = pd.read_csv("../../data/boston_traced_map_norm.csv")
df[:5]
```

	s	d	w
0	0	1	11
1	1	2	12
2	2	3	11
3	0	4	14
4	3	0	10

Apply Dijkstras:

```
class BostonGraph():
    """
    A class that creates a graph network to be used with Dijkstras Algorithm
    """

    def __init__(self, nodes):
        """
        A constructor that creates an instance of a graph using one
        argument for the number of nodes
        """
        self.nodes = set(range(1, nodes))
        self.edges = {}
        self.edge_distances = {}

    def add_new_edge(self, source_node, destination_node, distance):
        """
        A function that adds a new edge to the graph network
        @input source_node: The node from which the edge starts
        @input destination_node: The node from which the edge ends
        @input distance: The weight of the node that represents distance
        """
        self.helper_add_edge(source_node, destination_node, distance)
        self.helper_add_edge(destination_node, source_node, distance)

    def helper_add_edge(self, source_node, destination_node, distance):
        """
        A helper function that adds a new edge to the graph network
        @input source_node: The node from which the edge starts
        @input destination_node: The node from which the edge ends
        @input distance: The weight of the node that represents distance
        """
        self.edges.setdefault(source_node, [])
        self.edges[source_node].append(destination_node)
        self.edge_distances[(source_node, destination_node)] = distance
```

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

```python
def apply_dijkstra(Graph, source_node):
    """
    A function that applies Dijkstras algorithm to a given network

    @input Graph: The graph network that the algorithm should be applied to
    @input source_node: The node from which the edge starts
    """

    # Define the nodes that were visited, and the current node
    visited = {source_node: 0}
    current_node = source_node
    spec_path = {}

    # Set the nodes
    nodes = set(Graph.nodes)

    # Enter while condition to start
    while nodes:
        min_node = None
        # Iterate over the nodes to check if visited
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node

        if min_node is None:
            break

        nodes.remove(min_node)
        cur_wt = visited[min_node]

        # Iterate over the edges
        for edge in Graph.edges[min_node]:
            wt = cur_wt + Graph.edge_distances[(min_node, edge)]
            # Set if condition of visited not met
            if edge not in visited or wt < visited[edge]:
                visited[edge] = wt
                spec_path[edge] = min_node

    return visited, spec_path
```

```python
def create_new_route(graph, start, end):
    """
    Function that creates a route or path by applying Dijkstras
    """
    distances, paths = apply_dijkstra(graph, start)
    route = [end]

    # Enter while loop to add paths to route
    while end != start:
        route.append(paths[end])
        end = paths[end]

    # Inverse to correct for order
    route.reverse()

    return route

def print_full_path(graph, start,end):
    """
    Function that graphically shows the route
    """
    # Call the create new route function
    full_route = create_new_route(graph, start,end)

    # Print the route
    print(full_route)


if __name__ == '__main__':

    # Create new graph
    g = BostonGraph(34)

    # Iterate over CSV and populate graph
    for row in df.values:
        g.add_new_edge(row[0], row[1], row[2])

    # Print the final path:
    print_full_path(g, 1, 22)
```

```
[1, 2, 10, 9, 8, 18, 19, 20, 22]
```

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Code - Method 3

Import Libraries:

```
import networkx as nx
import matplotlib.pyplot as plt
import sys
import pandas as pd
from collections import defaultdict, deque
```

```
print(nx.__version__)
```

```
2.8.8
```

Import Data:

```
df = pd.read_csv("../data/boston_traced_map_norm.csv")
df[:5]
```

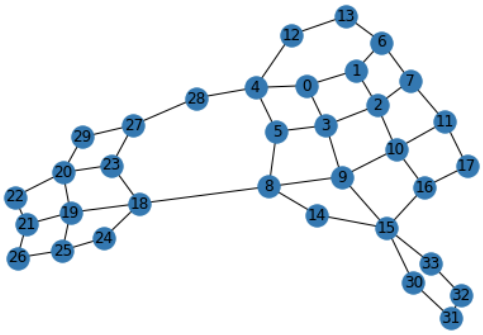
	s	d	w
0	0	1	11
1	1	2	12
2	2	3	11
3	0	4	14
4	3	0	10

Create Graph:

```
G = nx.Graph()
```

```
for row in df.values:
    G.add_edge(row[0],row[1], weight=row[2])
```

```
nx.draw(G, with_labels=True)
plt.show()
```



Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Apply TSP Algorithm:

```
tsp = nx.approximation.traveling_salesman_problem
```

```
tsp(G, nodes=[0, 20, 18, 9, 27])
```

```
[0, 4, 28, 27, 23, 20, 19, 18, 8, 9, 3, 0]
```

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Code - Method 4

```
import math
import copy

posroute= []
posroute2= []
disttravel= int
dist=0
targetdist=8
roaddict = {}
roadcondition ={"flood": .2, "ice":.7 , "snow": .8}
maxquality=0

class node:
        def __init__(self,name, xcord,ycord,adj):
                self.name = name
                self.xcord = xcord
                self.ycord = ycord
                self.adj = adj
        def buildroads(self):
                global roaddict
                # roaddict[self.name] = self.name + "-" +
                for ti in self.adj:
                        roadname = self.name + "-" + ti.name
                        roaddict[roadname] = 1
def updateroad (node1, node2, condition):
        roaddict.update({node1.name+"-"+node2.name : condition})
        roaddict.update({node2.name+"-"+node1.name : condition})
def quality1(currentnode,previous,dist):
        result= roaddict[currentnode.name+"-"+previous.name] * dist
        return result


def disttravel(currentnode,previous,travellist):
       global maxquality
       # print(posroute2)
       travellist.append(currentnode)
       # print(currentnode.name)
       tdist = 0
       tqual=0
       for k in range(len(travellist)-1):
               dist1 = distance(travellist[k+1].xcord,travellist[k+1].ycord,travellist[k].xcord,travellist[k].ycord)
               tdist = tdist +dist1
               quality = quality1(travellist[k+1],travellist[k],dist1)
               tqual = quality + tqual
       if tdist != 0:
               tqual = tqual/tdist
       else:
               tqual = 0
       newdist = distance(currentnode.xcord,currentnode.ycord,previous.xcord,previous.ycord)
       if(currentnode.name == travellist[0].name and targetdist-.2 <= tdist <= targetdist+2):
               print(tqual)
               if maxquality == 0:
                       maxquality= tqual
                       posroute2.append(travellist.copy())
               elif maxquality == tqual:
                       posroute2.append(travellist.copy())
               elif tqual > maxquality:
                       maxquality = tqual
                       posroute2.clear()
                       posroute2.append(travellist.copy())
               tqual = 0
       elif(tdist < targetdist):
               for i in currentnode.adj:
                       if(i.name != previous.name):
                              disttravel(i, currentnode, travellist)
                              if(len(travellist) >0):
                                      travellist.pop()
```

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

Code - Method 5

Reducing the graph by keeping a representation of the shortest distance between interesting sites and the source node

```
final_edges_df = pd.DataFrame(columns=["s","d","w"])
# create edges of interesting nodes and source
s = []
d = []
w = []

reduced_routes = {}

for i in range(len(final_nodes)):
  for j in range(i+1,len(final_nodes)):
    start = final_nodes.iloc[i]["Node"]
    end = final_nodes.iloc[j]["Node"]

    # Djikstra's algorithm to find routes
    route, distance = create_new_route(g, start, end)
    key = str(start) + "-" +  str(end)
    reduced_routes[key] = route
    s.append(start)
    d.append(end)
    w.append(distance)

final_edges_df["s"] = s
final_edges_df["d"] = d
final_edges_df["w"] = w
final_edges_df["route"] = len(final_edges_df)

# Sort edges ascending by distance
final_edges_df = final_edges_df.sort_values(by=["w"]).reset_index()[["s","d","w","route"]]
final_edges_df[:5]
```

Greedy algorithm for choosing the closest path from the current node:

Saleh Alkhalifa
Gabriel Cuchacovic
Nile Naim

```
# Greedy Approach
distance = 100
tolerance = 0.1*distance
current_distance = 0
current_node = source_node
current_route_index = 0
index = final_edges_df.index
edges_len = len(final_edges_df)
visited_nodes = [source_node]

# Greedy Approach
while True:
  before = current_distance
  for i in range(edges_len):
    edge = final_edges_df.iloc[i]

    if edge["s"] == current_node and edge["route"] == edges_len and edge["d"] not in visited_nodes \
    and current_distance + edge["w"] <= distance:
      final_edges_df.at[index[i], "route"] = current_route_index
      current_route_index += 1
      current_node = edge["d"]
      current_distance += edge["w"]
      visited_nodes.append(current_node)
      break
    elif edge["d"] == current_node and edge["route"] == edges_len and edge["s"] not in visited_nodes \
    and current_distance + edge["w"] <= distance:
      final_edges_df.at[index[i], "route"] = current_route_index
      current_route_index += 1
      current_node = edge["s"]
      current_distance += edge["w"]
      visited_nodes.append(current_node)
      break

  if before == current_distance:
    break
```