Application of Graph Neural Networks for Optimization and Design of Modern Subway Systems

Application of Graph Neural Networks for Optimization and Design of Modern Subway Systems

Andrew Jo[1], Daniel Min[2], Nathan Cardin[1]

[1] Harvard-Westlake Upper School, 3700 Coldwater Canyon Ave; [2] UCLA, Los Angeles, 90095

Abstract

Networks are a fundamental part of everyday life. Networks underlie critical infrastructures such as roads, railways, and power grids. Designing an optimal network however, even for two variables, is computationally challenging. In practice, multiple variables—such as cost, efficiency, and resilience—must be considered simultaneously. In this work, I analyze a deep learning approach using Graph Neural Networks (GNNs) for more efficient design of near-optimal networks. The current best method for determining global efficiency has a time complexity of $O(N^3)$. In this research project, I use the data from the General Transit Feed Specification (GTFS) to model subway systems in the United States as graphs. My results demonstrate that training GNNs can outperform conventional approaches by offering constant-time approximations of global efficiency. Small changes in my GNN training algorithm should also be applicable to other types of networks, such as roads and power grids. My results here provide the basis for a novel way of designing and analyzing complex networks.

Introduction

The field of graph theory has provided us a powerful and flexible tool through which we can analyze the networks we design. One graph-like structure that inspired this project was the development of slime molds, a yellow protist that seeks out nearby food and establishes multiple interweaving lines to transport nutrients across the colony. Biologists and computer scientists have analyzed slime mold growth in order to extract knowledge we can apply to man-made designs. For instance, we can analyze subway systems as graphs, where each station is a node, and a trip between stations is an edge.

While graphs can be measured via many metrics, often the two biggest ones are global efficiency and cost. Global efficiency is a measure of how quickly one can get from a random point on a graph to another. Cost keeps track of the combined length of all edges within a graph. Thus, optimal graph design is often a balance of trying to find as much efficiency within certain costs. However, there is one major problem with this analysis. The current best way to calculate global efficiency is via an algorithm, named Floyd-Warshall, with a time complexity of O(V^3). This means that the time it takes to calculate the global efficiency of a graph increases cubically with the number of nodes in the graph. This is manageable for computers when analyzing a graph with a few hundred or even thousand nodes, but it quickly grows to untenable proportions.

Fortunately, AI is very strong with regression, which is the process of estimating metrics based on input data. Graph Neural Networks can take in all the data from a graph as a couple simple matrices and learn to compute certain values at near instant speeds. While this sacrifices the complete accuracy of the original algorithm, these fast approximations can be a powerful tool for assessing multiple potential graphs for a subway, at which point you can narrow down a couple good candidates and calculate them with the original algorithm.

Materials

Language

The entirety of this project was written in Python and stored in a Jupyter notebook.

Databases

The Transitland API

Libraries

Library Name	Description of Use
Pandas	Read in GTFS Data and organize it in an easily accessible table format
NetworkX	Provides Graph class and allows for easy display of graphs
PyPlot	A quick way to graph data in different formats
Torch/Torch Geometric	Provides the framework for a GNN model
Optuna	Helps search through different hyperparameters in an efficient manner

Method

Acquiring Data

General Transit Feed Specification (GTFS) is a standardized data format designed for representing comprehensive information on many transit systems, including subways around the world. The Transitland API is a service that provides up-to-date GTFS data when prompted with an operator ID, all of which can be found by geographic location on the Transitland website. In order to systematically collect as many subways as possible, I took a copy of the html for Transitland website's page for all Californian operators and had my code isolate every unique operator id.

Making Graphs

GTFS data can come with a lot of files with varying amounts of data, but there are a couple data points that are required for every GTFS directory, so we'll only rely on those to make our graphs. The data I used can specifically be found in the stops.txt, trips.txt, and stop_times.txt files. All the data for our nodes can be found in stops.txt, which gives a unique stop_id for each stop alongside a longitude and latitude. While graph nodes don't need to be in a strict position in space, I chose to incorporate the longitude and latitude values as x,y coordinates to make the graphs resemble their real-world counterparts. Then, trips.txt gives a list of all trips and their trip_ids. However, to get a sense of what stops are in each trip, we need stop_times.txt. Every stop time is associated with a specific stop location along one of the trips, which is indicated by a matching stop_id and trip_id respectively. Every stop time also has a stop_sequence value, indicating that it is the nth stop of the trip. By sorting all the stop_sequence values from smallest to greatest, we know that each stop in the sequence connects directly to the next stop in the sequence, giving us all of the edges.

Preprocessing

Now that we have all our nodes and edges, there are a couple important decisions to make about our graph. First, we'll make it a weighted graph, meaning each edge has a value associated with it. In this case, that value is the physical distance between the two stations, calculated using the classic linear distance formula and treating longitude and latitude as coordinate values since the surface is flat enough. Second, I decided to make the graph undirected, despite the fact that some trips are one way and thus directed. This is a loss of information, but calculating global efficiency requires that every node be able to reach every other node, and this may not be possible if certain stops cannot return to previous stops. Lastly, I chose to filter out graphs that were too fractured. There were some operators where different groups of stops were detached from each other entirely, and while we can work around this issue, there were a couple egregious cases of tens or hundreds of subway stops that were supposedly connected to nothing. To me, this felt like an error in the GTFS data, or at the very least unusable data, so I chose to filter it out. To do so, I counted the number of nodes in each graph and divided that by the number of subgroups or subgraphs. This gave a subgraph density, which indicated how many nodes were in each group on average. I filtered out graphs with a subgraph density of 5 or less.

Graph Analysis

The most standard versions of the efficiency and cost functions on graphs are already very informative for our subway graphs, but lack normalization. For instance, we don't want larger graphs to be deemed less efficient purely because of their size.

The standard global efficiency function is defined as the average of all the inverse distances between every pair of points. For a graph with n nodes, you'd go through all pairs of i^{th}, j^{th} nodes, where i is not j, and divide that by the number of pairs of nodes.

$$E = \frac{1}{n(n-1)} * \Sigma \frac{1}{d_{ij}}$$

This works great for unweighted graphs, where every distance is considered 1, but can quickly take on a wide range of values in an unweighted graph with a variety of distances. To ensure that our efficiency value remains between 0 and 1, we can compare it to the ideal graph. This would be a version of the graph where every node has a direct path to every other node in the graph, making it perfectly efficient. Thus, our actual graph's efficiency will lay between 0 and the ideal graph's efficiency.

$$E_{glob}(G) = \frac{E(G)}{E(G^{ideal})}$$

Lastly, as mentioned earlier, some of our graphs are not fully connected and thus don't have valid distance values between groups. However, subways aren't always meant to be ridden from every single location to another, and people in real life will walk between nearby lines. Thus, we can simulate this commute by adding temporary edges between

subgraphs for the efficiency calculation, though these edges will be weighted higher than normal subway edges, as walking the same distance is considerably slower on foot.

A normalized cost function can be achieved similarly. The standard cost function is simply the sum of all the weights of edges in the graph. You can then compare that cost to what is called the greedy triangulation of the graph, which continues to add more and more edges until it can't anymore without creating intersections.

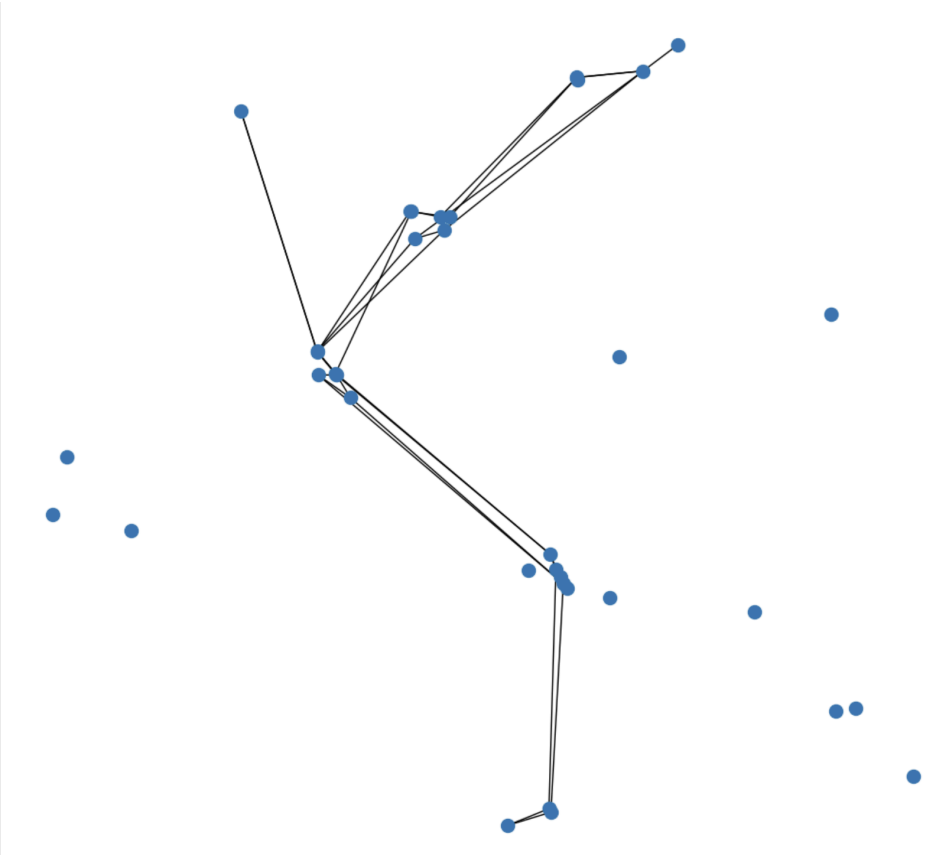
Graph Neural Network

Finally, we can use all our graphs and data to train and test a graph neural network. I chose to use the PyTorch Geometric library's Graph Convolutional Network model. This model can take in matrices that represent our graphs and adjust its parameters to accurately predict the target value. For our graphs, the model only needs to take in a matrix of all our nodes containing their longitudes and latitudes in each row, and one more matrix representing all pairs of nodes that have an edge between them. Finally, we can use Optuna to efficiently test different hyper-parameters such as the size of hidden layers and learning rate to arrive at a more accurate model and analyze its capabilities.

Results

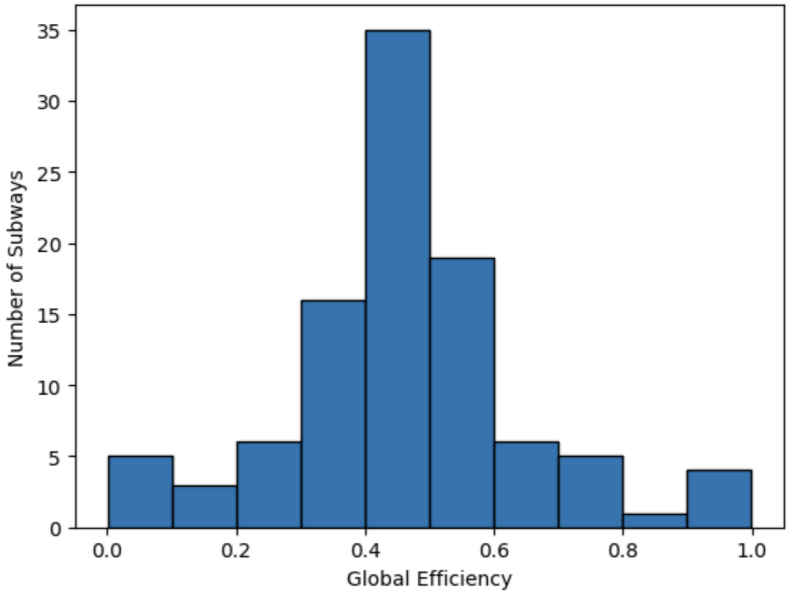
California Subway Analysis

As mentioned above, one major observation was that there appeared to be some incomplete data in the database. For example, the “f-9qf-calaveras~ca~us” operator:



While the central body looks like a reasonable subway system, all the extra nodes scattered around it would only be highly confusing to the algorithm and difficult to fix reasonably. Thus, graphs of this nature were removed.

California Subway Efficiency Distribution:

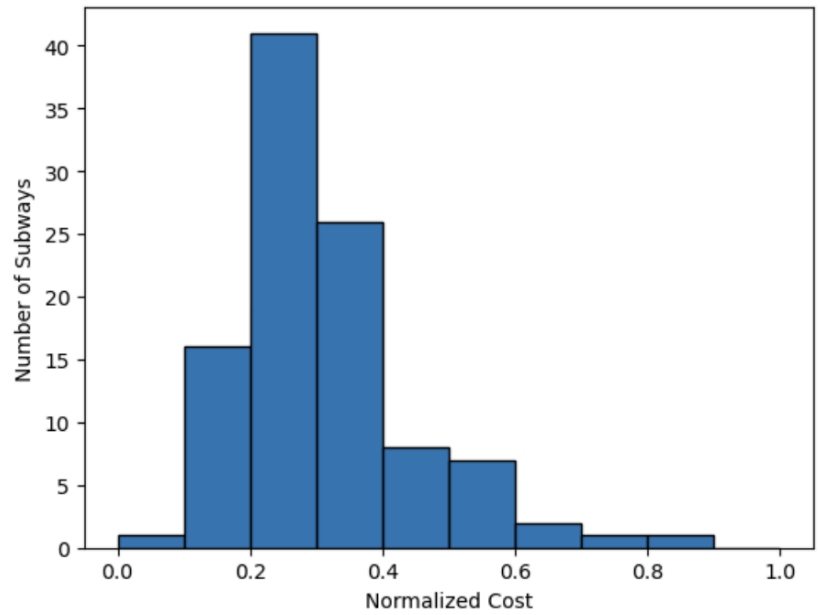


Average Efficiency: 0.46254

Median Efficiency: 0.44854

Standard Deviation: 0.18976

California Subway Cost Distribution:



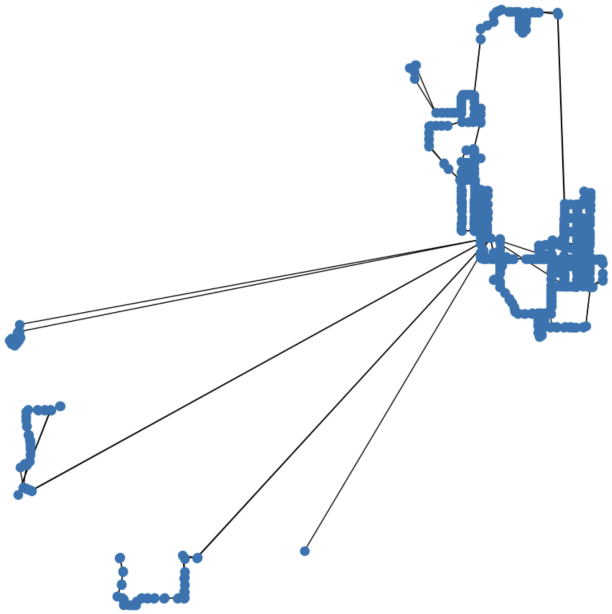
Average Cost: 0.46519

Median Cost: 0.42930

Standard Deviation: 0.15090

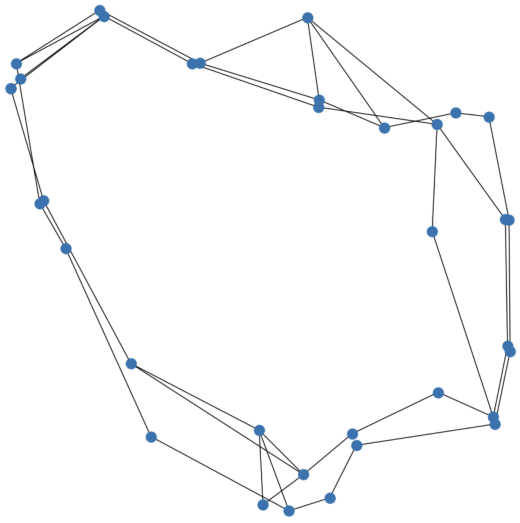
Least Efficient Graph “f-antelope~valley~transit~authority”:

Efficiency: 0.0003654931712
Cost: 0.4844595778
Graph:



Most Efficient Graph “f-9q9-ucsc~taps”:

Efficiency: 0.999933267
Cost: 0.7348672369
Graph:



The most and least efficient subways in California seem to demonstrate very different design philosophies. The most efficient graph from operator ”f-9q9-ucsc~taps” uses a loop, ensuring that no node is ever more than half a rotation away from another. Even cost wise, the graph is more costly than the average subway, but . Meanwhile, the least efficient subway “f-antelope~valley~transit~authority” appears to contain one major hub of subway stations and four smaller islands that it reaches out to include. While the graph is optimized around travel around the central hub, its biggest weakness is transportation from one of the smaller islands to another. However, perhaps these smaller areas simply have lower demand for subway travel, and optimization around the larger area was a reasonable design choice. It’s definitely worth further investigating the significance of certain nodes in relation to the density of the surrounding population.

Hyperparameter Optimization

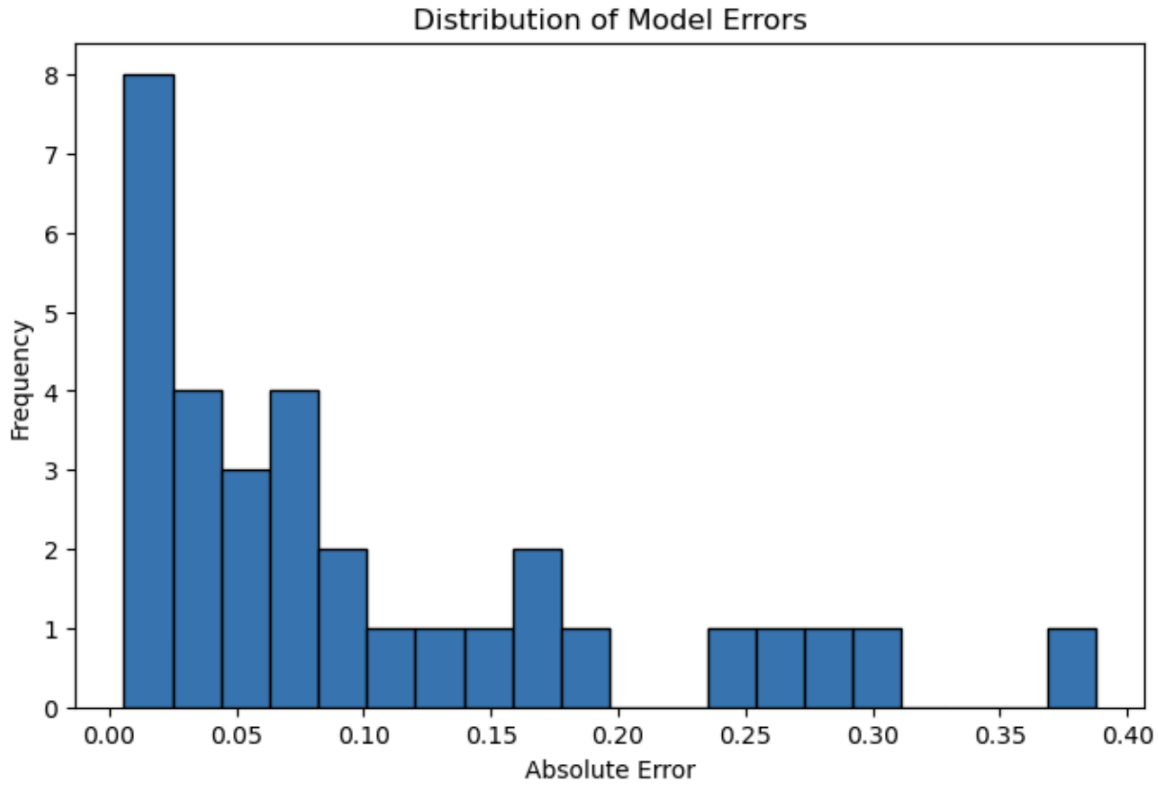
Random Seed: 42
Hidden Dimension Size: 93
Learning Rate = $5.09792 * 10^{-3}$

Epoch Training and Losses:

```
Epoch 1: Train Loss = 12.0504, Test Loss = 0.0150
Epoch 2: Train Loss = 0.0285, Test Loss = 0.0114
Epoch 3: Train Loss = 0.0410, Test Loss = 0.0119
Epoch 4: Train Loss = 0.0263, Test Loss = 0.0153
Epoch 5: Train Loss = 0.0264, Test Loss = 0.0142
Epoch 6: Train Loss = 0.0280, Test Loss = 0.0112
Epoch 7: Train Loss = 0.0370, Test Loss = 0.0371
Epoch 8: Train Loss = 0.0287, Test Loss = 0.0118
Epoch 9: Train Loss = 0.0174, Test Loss = 0.0100
Epoch 10: Train Loss = 0.0166, Test Loss = 0.0101
Epoch 11: Train Loss = 0.0166, Test Loss = 0.0102
Epoch 12: Train Loss = 0.0166, Test Loss = 0.0102
Epoch 13: Train Loss = 0.0166, Test Loss = 0.0102
Epoch 14: Train Loss = 0.0167, Test Loss = 0.0102
Epoch 15: Train Loss = 0.0167, Test Loss = 0.0103
Epoch 16: Train Loss = 0.0167, Test Loss = 0.0103
Epoch 17: Train Loss = 0.0167, Test Loss = 0.0103
Epoch 18: Train Loss = 0.0167, Test Loss = 0.0103
Epoch 19: Train Loss = 0.0167, Test Loss = 0.0103
Epoch 20: Train Loss = 0.0167, Test Loss = 0.0103
```

As expected with AI training, the initial epochs have the greatest loss as the model hasn't had much time to adjust to the data, but it gradually improves. It becomes optimal at the 9th epoch before it begins to overfit on the training data.

Model Accuracy



Average Error: 0.10193

Median Error: 0.06922

Standard Deviation: 0.09849

Overall, the model is fairly consistent in estimating graph efficiency within ± 0.10 of the actual value, though is prone to occasional big outliers. This can definitely be a big risk when analyzing massive sets of graphs, as the model may overestimate a potentially great design or overly favor a poor design. However, the high frequency of strong approximations is encouraging, as it demonstrates that the AI is capable of understanding the features of efficient graphs and has room to improve with more data with more precise features.

Conclusion

It's clear that the subways of California vary wildly in their efficiencies and costs, revealing room for these subways or future subways to be designed with a better balance. The Graph Neural Network already shows incredible promise for serving as a quick and accurate arbiter for a graph's efficiency, especially given the fact that it only used data from California's subway systems. Given the comprehensiveness of Transifteed's database, this same procedure can definitely be used to gather data from other US States or countries around the world to train the AI, granted that you have the computing power to handle large quantities of big subways, such as the ones in New York.

There is also a lot of room to expand upon the nuance of subway analysis by going back and reincorporating directed graphs, giving certain nodes higher priority based on population density around those areas, and adding extra costs for building through unfavorable terrain. The more accurately we can account for the strains on the subway system and people's demands of it, the more successfully we can integrate subway travel into more of the world. Of course, this graph analysis is incredibly flexible, and can also be a useful tool in the further development of all kinds of infrastructure for societies going forward.

One exciting application for this efficiency-calculating neural network is to use it as a critic AI in an actor-critic reinforcement learning algorithm. Critic AI models are a type of AI model that provides feedback to the actor AI, which is often attempting to learn how to make complex, optimal decisions. An AI model attempting to optimize graphs may have been restricted by the slow recalculation of efficiency during graph modification, but my neural network could provide instant feedback to the actor AI and allow it to efficiently test new potential configurations of the graph.

References

1. Atsushi Tero *et al.*, Rules for Biologically Inspired Adaptive Network Design. *Science* **327**, 439-442 (2010). DOI: 10.1126/science.1177894

2. Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., ... & Levine, S. (2018). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.

3. Khemani, B., Patil, S., Kotecha, K. *et al.* A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *J Big Data* **11**, 18 (2024). https://doi.org/10.1186/s40537-023-00876-4

4. Latora, V., & Marchiori, M. (2001). Efficient behavior of small-world networks. *Physical review letters*, *87*(19), 198701. https://doi.org/10.1103/PhysRevLett.87.198701

5. Latora, V., & Marchiori, M. (2002). Economic small-world behavior in weighted networks. *The European Physical Journal B - Condensed Matter and Complex Systems, 32*, 249-263.

6. Risald, A. E. Mirino and Suyoto, "Best routes selection using Dijkstra and Floyd-Warshall algorithm," *2017 11th International Conference on Information & Communication Technology and System (ICTS)*, Surabaya, Indonesia, 2017, pp. 155-158, doi: 10.1109/ICTS.2017.8265662. keywords: {Algorithm design and analysis;Routing;Prediction algorithms;Information and communication technology;Shortest path problem;Complex networks;Health services;Dijkstra's Algorithm;Floyd-Warshall Intelligent System Algorithm},

7. Rodrigue, J.-P., Comtois, C., & Slack, B. (2024). *The geography of transport systems* (6th ed.). Routledge. https://doi.org/10.4324/9781003343196