A Real-Time Application of Dijkstra's Algorithm

Parth Shisode
EE122 Sp. '24
parths21@berkeley.edu

ABSTRACT

Graphs serve as generalized ways to represent many real-world phenomena. This can include the flow of traffic on a network of city streets, transfer of data through a computer network, or even the spread of information from person to person. Shortest-path algorithms are among the most critical elements of graph theory; Dijkstra's algorithm is one of the most famous. However, Dijkstra's is meant to handle static graphs whose edges don't change. Meanwhile, for example, modeling traffic flow throughout a city would have traffic flow on streets certainly change throughout the course of a day. Given that graphs generalize real-world situations so well, I present a framework for creating a shortest-path algorithm which accounts for changing edge weights, utilizing ideas from Dijkstra's algorithm. This project also includes a Python notebook, aimed at individuals who learn more effectively through interactive examples.

PROBLEM STATEMENT & INTRODUCTION

Graphs are an incredibly useful generalization of real-life events; it makes sense for graphs which can realistically represent these phenomena to have edge weights that change over time. Often, this data has a pattern. Machine learning techniques are useful in this scenario, and they are utilized here. This algorithm attempts to answer the following: given historical data and real-time data describing edges' weights over time, how can we route an agent from a source to a destination with the least total weight?

For this project, we assume that we have access to samples of edge weights over time. This means that we can use a machine learning technique to draw out a prediction of an edge weight at a specific time. If we have enough historical data, and this data is fairly granular, curve-fitting is an appropriate choice [7].

However, especially in the real world, historical data may not accurately reflect the distribution of real-time data. If models are trained on data that doesn't reflect the situation at hand, then they're not effectively useless. Online prediction techniques don't rely on a large amount of historical data. Rather, they update themselves as real-time data continues to come in.

However, assuming that the historical data is an accurate representation of what the real-time data could look like, a model trained on this historical data is most likely going be a better predictor than an online technique. Therefore, it makes sense to continue using our well-trained model until we somehow realize that our real-time data strays from our historical data.

Graphs are defined slightly differently in this project as compared to the typical definition. Edges have two elements: edge lengths and edge rates. For example, if the edge rate is constant the time required to cross an edge would be the edge length divided by the edge rate. "Time" is an abstract concept here and we're not working with units like seconds or minutes. The goal of the shortest-path algorithm I aim to structure is specifically to shorten the total time required from a source to a destination. This change is enacted to more smoothly work with the examples that are present in the Python notebook that was constructed for this project. Also, it's common for many real-world scenarios to have "edges" that have lengths that remain constant while the traffic which flows among them is variable.

The Python notebook constructed is the primary medium of information for this project[1]. The style was taken from U.C. Berkeley's data science

[1] The Python notebook can be found at https://github.com/21ShisodeParth/dynamicdijkstras/blob/main/example.ipynb.

department and provides a similar introduction to the topics of this extended abstract in a more interactive form for those not preferring research-like formats.

Literature Review

Originally, when trying to introduce a way to improve Dijkstra's algorithm to handle edges that change, my first thought was to improve the algorithm itself. I certainly believed that dealing with a routing application of Dijkstra's algorithm meant that there were some qualities of routing and heuristics that I could implement which could improve the algorithm. However, introducing additional heuristics to Dijkstra's algorithm introduces a limitation to the search space of the algorithm and a change in behavior. In a study where a heuristic that described the Euclidean distance remaining to the destination, the search area became restricted to around ⅓ of the nodes which is typically reached via the regular Dijkstra's algorithm [6]. In an algorithm where the overarching goal is to find the absolute shortest path, while this implementation is an interesting approach it wasn't utilized.

Another study utilizes Dijkstra's algorithm to predict the frequencies that certain roads will be taken by food delivery employees. After these frequencies are calculated, an inverse reinforcement learning algorithm is trained to return optimal paths from a restaurant to a destination [8]. This application of Dijkstra's to solve a shortest path problem in a non-static environment is the most similar I've found to this project. However, in this case Dijkstra's is used to generate gradients for every path, and from there the shortest path problem is treated like a mathematical optimization problem. Meanwhile, the primary structure of Dijkstra's is preserved to solve the problem for this project. However, this study is what inspired me to integrate not only a way to utilize an online algorithm to observe real-time edge weights as they come but make the most of sampled edges (assuming we have the resources to collect these samples) by training a model on these points with the curve-fitting technique.

When deciding on curve-fitting (as an extension of linear regression) as a trained model and ARIMA as an online model, I did so based on a survey study comparing several machine learning methods and the improvement they yield when implementation into SDN routing. This survey study highlighted linear regression and ARIMA (alongside LSTMs) as especially effective machine learning methods to integrate into routing methodologies [1]. Linear regression integration was found to have the best management strategy and routing performance while being interpretable and meeting user requirements [9]. Meanwhile, ARIMA is effective in handling dynamic changes in policy due to it having a "short term memory" when it comes to creating future predictions [3]. I interpreted this fact as ARIMA being a suitable choice for a situation where we cannot use our curve-fitting model, and instead must work in a situation where we do not need to utilize a large volume of historical data. See more about this in the "Alternate Prediction" part of the *Implementation* section.

Implementation

I. Edge Rate Prediction

There are many methods of curve-fitting. The orthogonal least-squares fit (OLSF) method is used in this project.

$$x = (A^T A)^{-1} A^T b$$

Equation 1: Least-Squares Closed Form Solution

In this case referring to Eq. 1, matrix A would represent the timestamps, vector b would represent the edge rate values, and vector x represents the coefficients which are used to generate the predictions based on this method.

The maximum likelihood estimation is achieved by this curve-fitting technique, whose loss function ultimately seeks to reduce the distance between the provided points and the curve's prediction [5]. A polynomial function will be returned by OLSF for every edge present in the graph, returning the edge rate as a function of time.

A truncated version of the data used to generate the polynomial function is present in Table 1. Please note that this data was randomly generated for this project; a random polynomial was used as a base with Gaussian noise added to mask the true underlying function.

time	*source*	*destination*	*edge rate*
0.05	0	1	65.55
0.10	0	1	63.91
…			
0.05	1	6	29.64
0.10	1	6	31.40
…			

Table 1: Example of Edge Rate Historical Data

II. Modification to Dijkstra's Algorithm

The purpose of Dijkstra's algorithm is to determine the shortest path and total resources (time in our case) required to reach every node from a source node. Note that edge weights change over time; a critical element of this algorithm we're constructing is determining what the edge rate will be in the future when considering whether the edge should be traveled.

OLSF provides us with a polynomial function that returns the predicted edge rate as a function of time. To calculate the time at which an agent is predicted to complete crossing an edge from nodes A to B, we can solve this equation for x.

$$\int_{time\ at\ A}^{x} f(t)\,dt \;=\; path\ length$$

Equation 2: Equation for Arrival Time at Destination

III. Prediction Error Detection

Error detection is a straightforward element of this algorithm. We use the root mean square error (RMSE) to measure the difference between the observed edge rate values and the predicted edge rate values.

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

Equation 3: Root Mean Squared Error (RMSE)

As time goes on, these two values are collected, and if the RMSE exceeds a certain threshold we use our alternate technique rather than curve-fitting (see more in the next part of this section). There is no hard rule as to what a good threshold would be. This depends entirely on the use-case of and should be decided by those implementing this algorithm.

IV. Alternate Prediction

When choosing models to use in an online setting, ARMA (Autoregressive Moving Average) seems to be a good choice for this [2]. According to Anava et. al, it's been found that ARMA models approach a similar performance whether the setting is online or if all the data had been accessible. The Autoregressive Integrated Moving Average (ARIMA) model seems to be a more appropriate choice for this project. As compared to ARMA, ARIMA uses the differences between points rather than the values themselves to forecast future points; I care more about the differences between the points than the values themselves when it comes to generating predictions.

Since we're not using historical data, we must train on the real-time data which has been observed since the start of the time period. As more observations are collected over time, the online technique may improve its accuracy due to a larger training sample.

ALGORITHM

Please note that curve-fitting and ARIMA do not necessarily need to be the well-trained and online models of choice, respectively. These models generally perform well and were somewhat arbitrarily chosen since the premise of this project doesn't surround a specific real-life use case. The best choice of models would be determined through a combination of evaluation techniques for predictive models and domain knowledge specific to the use-case of this algorithm. An effective technique to determine which models/hyperparameters to use is cross-validation, which utilizes different subsets of the training data to test the effectiveness of each model [4].

For the pseudo-code for this algorithm, see *Appendix A*. Note that every single time an agent reaches a new node, the model-predicted edge rate for the edge which is supposed to be taken next is compared to the real observed edge rate. As described in the *Implementation* section, this is done to determine whether the predictions regarding the next edge to be taken align with the observed data. Making this comparison whenever an agent reaches a node could be computationally expensive. It might make more sense to only run this prediction/observed value comparison every k nodes traveled; k would depend on the use-case of this algorithm, specifically the size of the graph.

EXPERIMENT AND INSIGHTS

The example constructed for this project is described in this section. The adjacency matrix describes the length of each of the edges. The data for the edge rates are based off randomly generated polynomials (not known to me) and Gaussian noise to obscure the data pattern. Every edge with a positive edge length in the adjacency matrix exists.

```
[[0, 54, 0, 0, 99, 64, 74, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 85, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 50],
  [52, 65, 0, 0, 51, 81, 69, 0, 0, 0],
  [89, 0, 62, 0, 0, 0, 0, 90, 0, 0],
  [59, 0, 95, 77, 0, 0, 0, 99, 82, 0],
  [92, 54, 78, 0, 0, 0, 0, 0, 90, 98],
  [0, 0, 0, 0, 67, 0, 0, 0, 0, 91],
  [0, 0, 90, 0, 0, 0, 55, 0, 0, 0],
    [0, 0, 97, 0, 0, 61, 0, 0, 0, 0]]
```

Figure 1: Adjacency Matrix Describing Graph Length

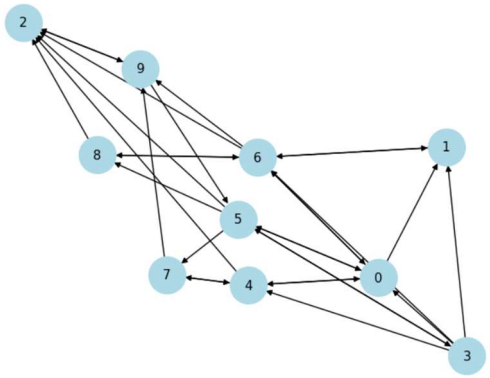
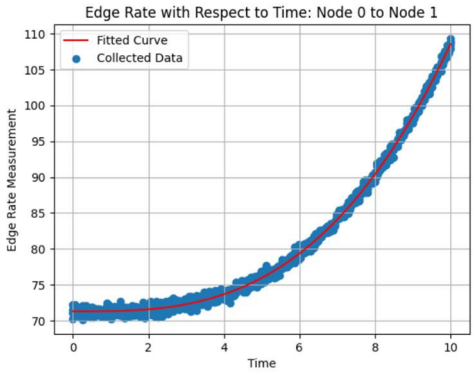


Figure 2: Graph Visual

Based off the historical data present for each of the existing edges (see example in Table 1), a polynomial was fit using the OLSF curve-fitting technique. Below is an example of the data and polynomial which was fit for Edge 0-1.

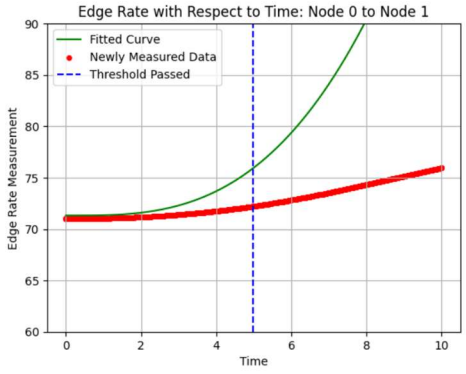


Graph 1: Edge 0-1 Data vs. Polynomial Fit

$$e(t) = 71.3170 + 0.00314t - 0.01865t^2 + 0.04431t^3 + 0.00087t^4 + 0.00003t^5$$

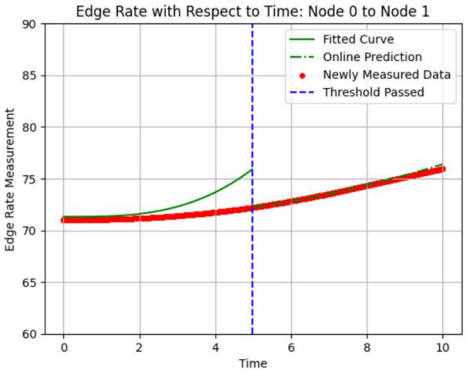
Equation 4: Edge Rate Formula for Edge 0-1

Now that we have the polynomial function generated from our historical data, we are prepared to deal with our new scenario in which we route an agent from a source to a destination in real-time. While it is typically expected that our historical data is similar to real-time data which is coming in, this is not always the case. The real-time data coming in for Edge 0-1 is different than that which we have fitted, as we see in Graph 2.



Graph 2: Real-time Data vs. Polynomial Fit

The real-time data seems to be similar at first, but it deviates from the polynomial curve generated from our historical data. As time goes on, the RMSE between the two curves slowly builds up until it crosses the threshold defined by the person running the experiment, roughly at time 5.



Graph 3: Real-time Data vs. Polynomial & ARIMA

Now, as soon as we reach time 5 and the RMSE threshold is crossed an emergency online technique is put in place. This example uses the ARIMA model, which is not trained on the historical data and instead only on the real-time data which has come in before time 5; this is the dashed curve.

When an agent is at a node, it is collecting the real-time edge rate, seeing whether it aligns with the edge rate predicted by the trained model, decides which model to use, then uses this model to run Dijkstra's algorithm. However, the real-time data could deviate from the model's prediction at any time. Therefore,

this process must be repeated every single time that an agent reaches a new node and has a new edge it is about to cross.

Dijkstra's algorithm was run on the graph below without considering how the real-time edge rates change. The agent simply saves the edge rates as soon as the journey from node 0 to the other nodes begins; the graph is treated statically regardless of the changing edge rates. The results of this can be seen in Figure 3.

```
From node 0 to node 0: [0]
From node 0 to node 1: [0, 1]
From node 0 to node 2: [0, 5, 2]
From node 0 to node 3: [0, 5, 3]
From node 0 to node 4: [0, 4]
From node 0 to node 5: [0, 5]
From node 0 to node 6: [0, 6]
From node 0 to node 7: [0, 5, 7]
From node 0 to node 8: [0, 5, 8]
From node 0 to node 9: [0, 5, 2, 9]

Time from Source to Destination
Node 0 to node 0: 0
Node 0 to node 1: 0.7571823536438264
Node 0 to node 2: 1.9573725040496626
Node 0 to node 3: 2.32347182420962
Node 0 to node 4: 1.2367786178281757
Node 0 to node 5: 0.8913511105433973
Node 0 to node 6: 0.8304514027749902
Node 0 to node 7: 2.7812126187042603
Node 0 to node 8: 2.152278489029063
Node 0 to node 9: 2.4582089608057105
```

Figure 3: Shortest Paths & Times for Node 0 with Dijkstra's Algorithm

Now, we consider the variable edge weights in our algorithm. This includes integrating the machine learning models discussed in the *Implementation* section into our modified version of Dijkstra's algorithm (see *Appendix A*).

```
Shortest Paths
Node 0 to node 0: [0]
Node 0 to node 1: [0, 1]
Node 0 to node 2: [0, 5, 2]
Node 0 to node 3: [0, 5, 3]
Node 0 to node 4: [0, 4]
Node 0 to node 5: [0, 5]
Node 0 to node 6: [0, 6]
Node 0 to node 7: [0, 5, 7]
Node 0 to node 8: [0, 6, 8]
Node 0 to node 9: [0, 6, 9]

Time from Source to Destination
Node 0 to node 0: 0
Node 0 to node 1: 0.7571570870587738
Node 0 to node 2: 2.84951307543477
Node 0 to node 3: 3.2006987543745753
Node 0 to node 4: 1.2355143096647345
Node 0 to node 5: 0.8910557437049846
Node 0 to node 6: 0.8310243245592757
Node 0 to node 7: 3.656658348750324
Node 0 to node 8: 3.0202997867879366
Node 0 to node 9: 4.220361737691761
```

Figure 3: Shortest Paths & Times for Node 0 with Real-Time Application of Dijkstra's Algorithm

There are two major differences in the running of the two algorithms. The static application of Dijkstra's algorithm which did not consider real-time data and did not use machine learning to predict future edge rates takes different routes from node 0 to nodes 8 and 9; see Figure 3. The second difference is in the times required to reach each node from node 0. Many of the times are different in the real-time application of Dijkstra's algorithm as compared to the static Dijkstra's. The static Dijkstra's algorithm fails to consider that edge rates change over time and makes the source-to-destination calculations based off the static edge rates originally measured. The real-time application of Dijkstra's does consider the real-time data and factors this into its calculations. In this example, based on the parameters passed into the random data generation, it seems that the edge rates slow down over time for many of the edges.

The experimental example doesn't include the element of the overhead required to re-run Dijkstra's at every node given that the model-predicted edge values are different from the real-time observed values. This example only aims to find the true shortest path and doesn't consider computational resources at all. Meanwhile, in an industry application this is a very real limiting factor. Additionally, this example doesn't include a real data stream and doesn't work with real time. Time is an abstract concept for this example and the data stream was just a function which returned a value when specifying an edge and the current time.

A framework for creating an algorithm combining principles of Dijkstra's algorithm, well-trained models, and online models has been created to deal with a real-time situation where we have historical data. Given the limitations, I'm satisfied with the project's results; this example shows that the algorithm can work. With the right domain knowledge and machine learning knowledge (mine is elementary compared to industry experts), this approach to real-time routing could have powerful implications.

ACKNOWLEDGEMENTS

My classmates in EE122 have put great effort into their own projects and taught me a lot about advances in the field of networking. I'd also like to thank Professor Shyam Parekh and the course staff of EE122 for their consistent willingness to help with the course and with this project.

REFERENCES

[1] R. Amin, E. Rojas, A. Aqdus, S. Ramzan, D. Casillas-Perez and J. M. Arco, "A Survey on Machine Learning Techniques for Routing Optimization in SDN," *IEEE Access vol. 9*, 2021, pp. 104582-10461

[2] O. Anava et. Al, "Online Learning for Time Series Prediction," *Proceedings of the 26th Annual Conference on Learning Theory*, 2013, pp. 172-184.

[3] F. Benamrane, M. Ali, D. K. Luong, Y. Hu, J. Li, and K. Abdo, ''Bandwidth management in avionic networks based on SDN paradigm and ML techniques,'' *Proc. IEEE/AIAA 38th Digit. Avionics Syst. Conf. (DASC),* Sep. 2019, pp. 1–9.

[4] D. Berrar, "Cross-validation," 2019

[5] N. Chernov and C. Lesort,
"Statistical efficiency of curve fitting algorithms,"
Computational Statistics & Data Analysis,
Volume 47, Issue 4, 2004, pp. 713-728.

[6] D. Fan and P. Shi, "Improvement of Dijkstra's algorithm and its application in route planning," *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, Yantai, China, 2010, pp. 1901-1904.

[7] R. de Levie. "Curve Fitting with Least Squares." *Critical Reviews in Analytical Chemistry* 30, 2000, pp. 59–74.

[8] Liu, Shan, et al. "Integrating Dijkstra's Algorithm into Deep Inverse Reinforcement Learning for Food Delivery Route Planning." *Transportation Research Part E: Logistics and Transportation Review*, vol. 142, 2020.

[9] L. Wang and D. T. Delaney, ''QoE oriented cognitive network based on machine learning and SDN,'' *Proc. IEEE 11th Int. Conf. Commun. Softw. Netw. (ICCSN)*, 2019, pp. 678–681.

APPENDIX A

```
threshold = …
real-time data = […]
predicted data = […]
edge models = […]

for each edge
    edge models[current edge] = curve_fit(historical data[current edge])

def dynamic dijkstra(model)
    times, shortest paths = …
    heap queue = …

    while heap queue not empty
        current time, current node = heap queue.pop()
        if time > times[node]
            continue

        for each neighbor
            time away = model.predict(time, source, neighbor)
            potential time = time + time away
            if potential_time < times[n]
                times[n] = potential time
                shortest paths[neighbor] = shortest paths[node] + i
                heap queue.push(potential time, n)
    return shortest paths

as time increases
    if agent arrived at a node
        next edge = dynamic dijkstra(model)[current node][upcoming edge]

        predicted edge weight = model.predict(current time, next edge)
        append(predicted data, predicted edge weight)
        observed edge weight = DATASTREAM.edge_weight(next edge)
        append(real-time data, observed edge weight)

        error = RMSE(predicted data, real-time data)
        if error > threshold
            model = ARIMA.fit(real-time data)
```