Time-dependent and Privacy-Preserving Decentralized Routing using Federated Learning

By

Chinmaya Samal

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

August 9, 2019

Nashville, Tennessee

ACKNOWLEDGMENTS

I am grateful to all those who helped me complete this work, especially my advisor Dr. Abhishek Dubey. He not only helped me by giving the right direction and vision to this research work but also opened my eyes to the bigger picture of this entire area of research. I appreciate his constant encouragement and guidance, which has made a significant contribution to the completion of this thesis. I would like to thank Dr. Aniruddha Gokhale for serving on the committee of my thesis. He has offered helpful guidance on my research in classes and projects in the past few years.

I would like to express my special thanks to my parents, Mr. Muralidhar Samal and Mrs. Swadhini Samal for their sacrificial love and best wishes. This research could not have been possible without the support of my friends, Geoffrey Pettet, Michael Wilbur, Sanchita Basak, Shreyas Ramakrishna, and Scott Eisele, who directly encouraged and supported me in research and writing.

Finally, I would like to acknowledge the research funding from National Science Foundation (NSF), Vanderbilt Initiative in Smart City Operations and Research, a trans-institutional initiative funded by the Vanderbilt University, for supporting the various research projects in which I have participated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

I.1 Problem Overview

Cities are evolving at a rapid pace. Over half the world's population currently lives in urban areas. By 2050, that number is expected to jump to 70%, according to the Population Division report from the UNs Department of Economic and Social Affairs (1). Along with a growing population, new challenges are emerging as an increase in housing density, population, and traffic can cause inefficiencies in services provided by the city. Managing and responding to urban incidents such as traffic accidents, fire and crime are fundamental challenges faced by a city, where any inefficiency can lead to loss of lives and erode the trust in the system.

The problem of dispatching emergency responders involves (a) a model that determines which responder to dispatch given incident location, time and (b) finding a route with minimum travel time to reach the incident location. In our prior research (2), we focused on the former. In this thesis, we focus on solving the latter.

Cities use third party services (3) to avail the emergency dispatch service and hence it comes with a cost that may not be affordable to cities with a limited budget. Furthermore, state of the art solutions (4; 5; 6) for route planning assumes a centralized approach, where a travel time model coupled with the shortest path routing algorithm finds a route for an emergency responder. One of the problems with this approach is that if there is a communication failure due to natural disasters, then the responders will fail to get a route plan and thus could not reach incident locations in time, which might result in loss of precious lives.

Cities are looking to improve everything from infrastructure to connectivity by developing technology to help cities efficiently provide necessary services. Hence they are investing in computation resources called Road Side Units (RSUs) that are strategically placed at certain roads, highways and can wirelessly communicate with vehicles. There is a need for a decentralized approach that can utilize a set of RSUs distributed in a city, to find a route plan for the emergency responders. As everyday devices are becoming more powerful (7), such an approach can effectively tackle the urban routing problem by harnessing the device resources and will particularly help cities with a limited budget and network coverage, provide self-sustaining mobility services for its residents while still preserving their privacy.

I.2 Open Challenges

Route planning is a well-studied topic in large part due to its practical relevance in real-world applications. The process of route planning often involves a human, which initiates a request to find a feasible route from origin to the destination which satisfies user preferences. State of the art solutions for route planning in a time-dependent network involves a centralized approach, where parallelization of the search algorithm uses a shared memory model. Hence, its deployment is limited to multiprocessing environment such as in a data center, where it is assumed that a shared memory allows a constant time direct communication between each pair of processors. It is not well suited for a distributed setting, which is prone to communication failures and can incur higher response times due to network latency. Furthermore, storing and using location data of users raises privacy concerns. We discuss this in more detail with a literature review in Section II.

I.3 Our Contributions

The goal of this thesis is to develop a routing algorithm that can efficiently search time-dependent network in a distributed manner and to develop a decentralized architecture where individual RSUs which have limited resources can operate in an environment with intermittent network connectivity. To this end, we make the following contributions in this thesis:-

1. We leverage recent advances in Federated learning (8) to collaboratively learn shared prediction models online while keeping all the training data on the device, thus preserving privacy.

2. We develop a resilient, decentralized approach for route planning in a time-dependent network, that leverage the models learned in the RSUs.

3. Finally, we show the effectiveness of our approach and provide analysis using a case study from the Metropolitan Nashville area. Our approach uses 1.2 - 6 times less memory per compute node than the central approach. Our approach has a moderately higher query response time than the central server approach, but our decentralized architecture makes it resilient in case of system failures and has high scalability.

Please note that while we discuss our approach with an example for emergency dispatch vehicles, our approach is still applicable for any vehicle and compute nodes can be mobile too.

I.4 Organization

The remainder of this thesis is organized as follows. We define basic notion for graphs including time-dependent network, discuss state of the art solutions for centralized route planning and its limitations in Chapter II. In Chapter III we present our decentralized architecture for route planning and discuss each component of the architecture and how they coordinate with each other to plan routes. We then evaluate our decentralized architecture with a case study for the Nashville metropolitan region in Chapter IV. In the same chapter, we first discuss our experiment setup and the data used in our experiments, then we evaluate different prediction models used in our work and finally, we evaluate our approach and compare it with state of the art solutions for centralized route planning. We then conclude the thesis, suggest avenues of further investigation in Chapter V. The source code is available on Github[1]

[1] https://git.isis.vanderbilt.edu/dabhishe/decentralized-routing

CHAPTER II

Background and Related work

In this section, we first develop basic notations which will be used throughout the work. Then we discuss the state of the art solutions for centralized route planning and its limitations in a decentralized setting. Table II.1 summarizes the symbols we used throughout this paper.

II.1 Graph fundamentals

Graphs: A graph is a tuple $G = (V, E)$ consisting of a finite set of vertices V and edges $E \subseteq V \times V$. There is an edge from $u \in V$ to $v \in V$ if and only if $(u, v) \in E$. Note that we use the terms graph and network interchangeably. We assume that our graph G is *directed*, i.e each edge $e \in E$ has a direction.

Edge weights: A graph is said to be weighted when a numerical label (i.e. weight) is assigned to each of its edges. For instance, there might be a cost involved in traveling from a vertex to one of its neighbors, in which case the weight assigned to the corresponding edge can represent such a cost. Let this cost be travel time and $T(e)$ be the travel time function which depends only on edge. Since $T(e)$ is constant over Π, where Π depicts a set of time points or time-period (seconds, minutes or hours of a day), $T(e)$ is called time-independent (9).

Routes: In time-independent graphs, a route query is defined by a tuple (s, d), where $s \in V$ is the source and $d \in V$ is the destination. Finding the shortest path or route R for the query (s, d) involves finding a sequence of edges $[e_1, e_2, \cdots, e_n]$ from source s to reach destination v, such that $e_i \in E$ holds. Note that we use the terms path and route interchangeably. The length of the route $len(R)$ is defined as the sum of weights for each edge in route, such that $len(R) = \sum_{e \in R} T(e)$.

Table II.1: List of symbols

Symbol	Description
$\mathbb{R}$	Real Numbers
$\mathbb{N}_0$	Natual numbers
G	Static graph, $G = (V, E)$
V	Set of network vertices
E	Set of network edges
τ_i	Actual time interval i of a day
$\hat{\tau}_i$	Estimated time interval i of a day
Π	Set of time points or time-period (seconds, minutes or hours of a day)
RSU_i	Road Side Unit i
R	Directed path from source vertex $s \in V$ to destination vertex $d \in V$, at time interval from source τ_s
R^s_d	Partial route from source vertex $s \in V$ to destination vertex $d \in V$, at time interval from source τ_s
$len(R)$	Travel time of the route R
G_τ	State of time-dependent graph (V, E) at time τ
V_τ	Set of vertices at at time interval τ
E_τ	Set of edges at at time interval τ
T	Travel time function
$\hat{T}$	Travel time predictor
$\hat{E}$	Equivalent Grid Routing predictor
SP	Shortest path algorithm that uses Travel time predictor $\hat{T}$ to find route with minimum travel time
g_i	Grid i
G^i	Subgraph whose each vertices and edges maps to grid i
$\hat{t}^u_v$	Estimated travel time from vertex $u \in V$ to vertex $v \in V$ using Travel time predictor $\hat{T}$

II.2 Transportation network as a Graph

Graphs: Transportation network can be realized as a time-dependent graph (9) since state of the network changes over time. Let $G_\tau = (V_\tau, E_\tau)$ be the time-dependent, directed graph, where V_τ is the set of vertices, $E_\tau \subseteq V_\tau \times V_\tau$ the set of edges of a road network at time interval τ. There is an edge from $u \in V_\tau$ to $v \in V_\tau$, if and only if $(u, v) \in E_\tau$.

Edge weights: Since the graph G_τ is time-dependent, the travel time on an edge $e \in E_\tau$ varies with time. All edges in a transportation network is weighted by periodic time-dependent travel time function $T(e, \tau) : \Pi \rightarrow \mathbb{N}_0$ where Π depicts a set of time points or time-period (seconds, minutes or hours of a day).

The function $T(e, \tau)$ is impacted by the routes taken by all the vehicles in the road network or graph G_τ and often can be modeled as a latency function (10). It can be learned from historical states of the network $\{G_0, G_1, ..., G_{\tau-1}\}$ and the current state of the network G_τ, to get expected travel times for time intervals $\{\tau+1, \tau+2, \cdots, \tau+f\}$, where f is the number of time intervals in future. To differentiate this from the actual travel time function, we denote the learned travel time function as $\hat{T}(e,\tau)$. The accuracy of estimates from $\hat{T}$ can result in inefficient route plan given as response by routers and thus affect the network congestion.

Routes: In time-dependent graphs, the shortest path depends on the departure time at the source node. Hence the route query is defined by a tuple (s,d,τ_s), where $s \in V_{\tau_s}$ is the source, $d \in V_{\tau_d}$ is the destination, τ_s is the departure time from s and τ_d is the arrival time at destination d. This might result in shortest paths of different length for different departure times or even a completely different route. In contrast to the time-independent graph, here the directed route R for the query (s,d,τ_s) involves finding a sequence of edges along the time $[(e_1,\tau_1),(e_2,\tau_2),\cdots,(e_n,\tau_n)]$ from source s at time τ_s to reach destination d at time τ_d. The length of the route $len(R)$, in time-dependent graph is defined as the sum of time-dependent weights function $T(e,\tau)$ for each edge in route, such that $len(R) = \sum_{(e,\tau)\in R} T(e,\tau)$.

II.3 Centralized Route Planning Architecture

In the context of emergency dispatch service, the goal is to minimize the variance in the operational delay between the time incidents are reported and when responders arrive on the scene. A critical component of this system is to find a route with minimum travel time. The process of route planning often involves an emergency responder or vehicle, which initiates a request to find a feasible route from origin to a destination which satisfies its preferences. State of the art solutions (4; 5; 6) for route planning assumes a centralized approach, where a central server or a set of servers in a data center manage network setup, handle real-time network updates, learn travel time models that aid a central router in finding shortest path

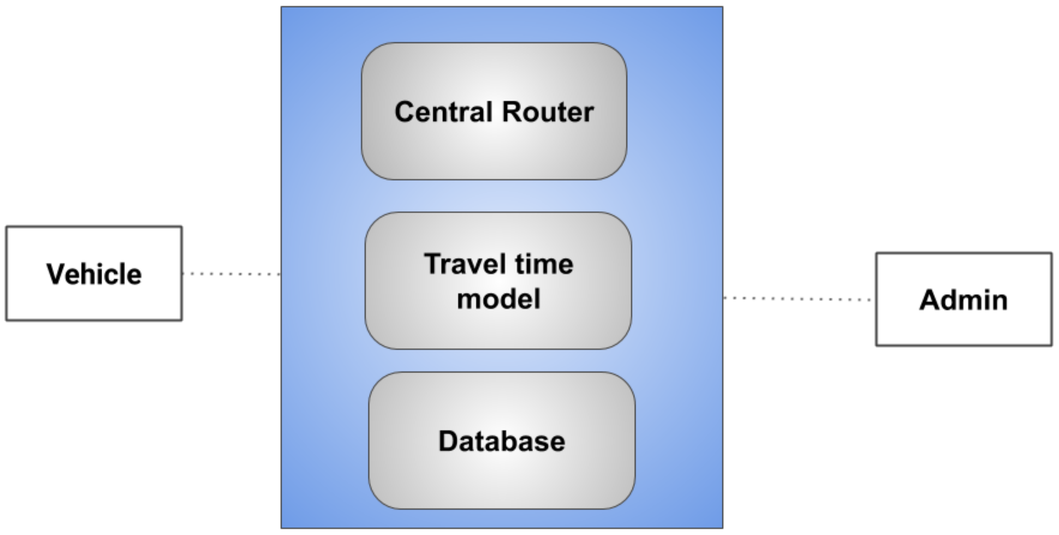

Figure II.1: Centralized architecture for route planning

queries. The central approach for route planning reduces network latency and also makes it efficient since all the data management can be done centrally and can be horizontally scaled if more resources are needed. Figure II.1 shows typical architecture for centralized route planning used in the state of the art solutions. We give a brief description of entities involved in the centralized architecture below:

1. **Vehicle**: It acts as a client to the *Central Server*. Multiple vehicles can be present in the system and each of them periodically sends location data to the *Central Server* in form of (e, τ, t), where e is the edge traversed, τ is the time interval of the day and t is the time taken to traverse e. It also sends timed routing queries to the *Central Server* in form of (s, d, τ_s), where s is the source vertex, d is the destination vertex and τ_s is the departure time from s.

2. **Admin**: It is responsible for adding the full time-dependent road network $G = (V, E)$ to the *Central Server* and handling real-time updates to the network G. It also designs and deploy travel time model $T(e, \tau)$ that helps the *Central Router* in finding a route

with minimum travel time.

3. **Central Server**: It abstracts a compute node and can scale its resources horizontally on demand. It consists of the following components:

   (a) **Database**: This component is responsible for storing data provided by *Vehicle* and *Admin*. It periodically collects location data from multiple vehicles and are stored as a set $\{(e_1,\tau_1,t_1),(e_2,\tau_2,t_2),\cdots,(e_n,\tau_n,t_n)\}$, where e_i is the edge traversed, τ_i is the time interval of the day. It also stores and updates network data $G=(V,E)$ provided by *Admin*.

   (b) **Travel time model**: Location data collected from multiple vehicles as discussed earlier, coupled with the road network, are used to train a data-driven model $\hat{T}$.

   (c) **Central Router**: It is responsible for finding a route given a query (s,d,τ_s) from *Vehicle*, where $s \in V$ is the source, $d \in V$ is the destination, τ_s is the departure time from s. It uses network stored in the *Database* and the *Travel time model* to find a route with minimum travel time. Inaccuracy in *Travel time model* can lead to inefficient route plans, i.e routes with increased or decreased travel times.

II.4 Related work for Route Planning

Given a timed query (s,d,τ_s), where s is the source vertex, d is the destination vertex and τ_s is the departure time from s, the goal of *Central Router* is to find a route with minimum travel time. To this end, we draw on two bodies of work in routing algorithms that heavily influence the central architecture for route planning. The first is the literature on single server routing where it is assumed that the network resides entirely in a physical node or server, to which any client sends routing queries to. The second is the literature on parallel routing where the network still resides in a physical node but is partitioned to multiple processes and the routing queries are processed concurrently using multiple processes.

Single server approach: Dijkstra (11) and Bellman and Ford (12; 13) proposed some of the first algorithms to solve the routing problem in a single server. Although these algorithms are quite old, many advanced route planning algorithms that exist today are variants of them. While these algorithms compute optimal shortest paths, they are too slow to process real-world data sets such as those deriving from large-scale road networks. To address this issue, there are many techniques aimed at speeding up these algorithms. Such techniques often are based on clever heuristics that accelerate the basic shortest paths algorithms by reducing their search space. Bi-directional search (14; 6), e.g., not only computes the shortest path from the source s to the target t, but simultaneously computes the shortest path from t to s on the backward graph. Goal-directed search such as A^* (15) uses other heuristics to guide the search. Goldberg et al. proposed the *ALT* approach in which they enhance A^* by introducing landmarks to compute feasible potential functions using the triangle inequality (6; 16). In other work, contraction techniques are used to speed-up the shortest path computation; e.g., *highway hierarchies* (17; 4) exploits the hierarchical in road networks, while *contraction hierarchies* (5) is based on contracting the graph.

Parallel approach: In this approach, a full road network is partitioned into multiple processes and the edge expansion proceeds similarly to Dijkstra. But here the difference is that unlike Dijkstra, the *priority queue* is based on a shared memory model where it is assumed that shared memory allows a constant time direct communication between each pair of processors (18; 19). This *parallel priority queue* supports simultaneous insertion and simultaneous decrease key of an arbitrary sequence of elements ordered according to key, in addition to find-minimum and single element delete operations (18; 20). Techniques to parallelize advanced routing algorithms such as *contraction hierarchies* discussed earlier, is only limited to the pre-processing step where the contraction of nodes can be done in parallel (21). Sets of nodes which can be contracted in parallel are iteratively found. By restricting the nodes to be contracted in each iteration node, contraction is done in parallel.

II.5 Limitations of Centralized Route Planning

As discussed earlier, our goal is to utilize a set of RSUs distributed in a city, to find a route plan for the emergency responders. The RSUs are resource-constrained and operate in an environment with intermittent network connectivity. We need a decentralized approach for route planning using these RSUs. However, the centralized architecture discussed above have limitations in a decentralized setting.

Some of the prior work discussed earlier, for parallel route planning (18; 19; 20), assume that the graph network has static weights, which doesn't hold in real transportation network where traffic congestion changes with time. In a time-dependent network, edge expansion depends on arrival/departure time at each edge, hence it needs to proceed sequentially. So, multiple processors cannot start expanding edges simultaneously. There are some approaches (21) which model the time-dependent nature of the network but the parallelization is only limited to the *pre-processing* phase and not during real-time query.

Furthermore, all of these approaches use a parallel shared memory model where an assumption is made that the shared memory allows a constant time direct communication between each pair of processors. This holds in a multiprocessing system and possibly in a data center, but this assumption is not realistic in a decentralized setting where nodes have intermittent connectivity and can fail anytime.

CHAPTER III

Decentralized Route Planner

In this chapter, we first give a brief overview of our decentralized architecture for route planning. Each component in our architecture can be better explained by their interactions with other components and the functionality they provide as a result of it. So, we first explain how the network is set up, then we discuss how data is collected, then we discuss in detail different prediction models we use to aid our decentralized route planner, then we discuss decentralized planning algorithm and finally we discuss how failures are handled in our architecture.

III.1 Architecture

Figure III.1 shows our decentralized architecture for route planning. Here are the functionalities of various entities in the system:

1. **Road Side Unit (RSU)**: This component abstracts a compute node that is installed by city planners near roads and highways. These nodes are assumed to have computational resources equal to those of Raspberry Pis or similar and are assumed to have intermittent network connectivity. It is responsible for preconfigured geographic region and aggregates real-time location data from the vehicles in the region configured. It also collects the trip data of vehicles. It facilitates the collaborative learning of shared prediction models while the data never leaves. We discuss more about the data collection and prediction models in Section III.2 and Section III.4 respectively. Finally, these prediction models are used by the decentralized route planning algorithm discussed in Section III.5.

2. **Central Server**: This component also abstracts a compute node, but it is assumed to have more resource available that can scale horizontally on demand. It is also

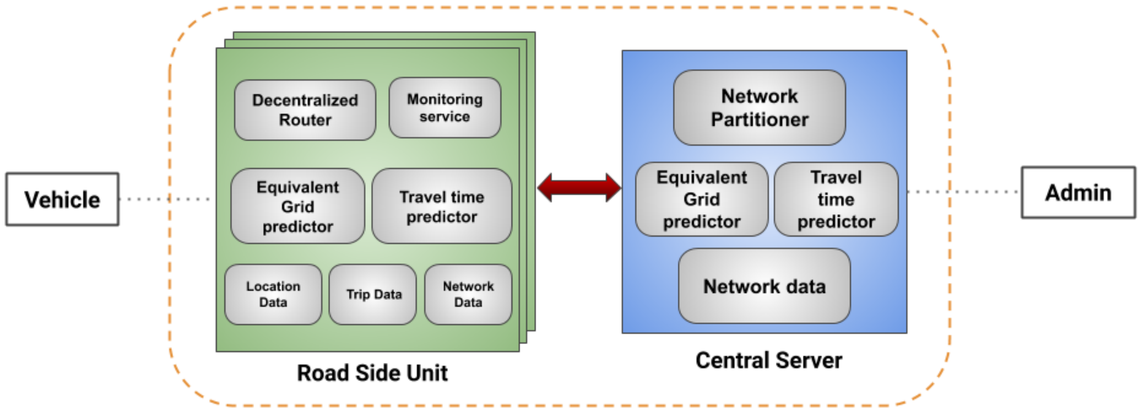

Figure III.1: Decentralized architecture for route planning

assumed to be a reliable backup node and is a central location for all the RSU to get the resources they need to function properly. We will discuss more the functionality of each component it contains, in the next sections.

3. **Vehicle**: It has same functionality as what discussed in Section II.3. The only difference here is that it sends real-time location data to the nearest RSU instead of a *Central Server* in a centralized architecture. Similarly, it also sends timed queries for route plans to the nearest RSU.

4. **Admin**: It has same functionality as what discussed in in Section II.3. The difference lies in the deployment of the model. In a centralized architecture, the inference was done on the *Central Server*, while in our approach, the inference is done in each RSUs where the data resides. We will discuss more this in the next sections.

This is different from the centralized architecture we discussed in Section II.3, in that most of the functionalities such as the route planning, learning prediction models and storing real-time data are now managed by RSU.

III.2 Data collection

Following types of data are stored in RSUs:

1. **Location data**: This data is periodically collected from from multiple vehicles and are stored as a set $\{(e_1, \tau_1, t_1), (e_2, \tau_2, t_2), \cdots, (e_n, \tau_n, t_n)\}$, where e_i is the edge traversed, τ_i is the time interval of the day. This data is used by the *Travel time predictor*, as discussed later in Section III.4.1.

2. **Trip data**: This data consists of a set of route plans $\{R_1, R_2, \cdots, R_n\}$ where each route is a sequence of edges along the time $R_i = [(e_1, \tau_1), (e_2, \tau_2), \cdots, (e_p, \tau_p)]$, as discussed earlier. The route is returned by *Decentralized Router* as a response for a user query (s, d, τ_s), where s is the source, d is the destination, τ_s is the departure time from s. This data is used by the *Equivalent Grid Routing predictor* predictor, as discussed later in Section III.4.2.

3. **Network data**: Network data $G = (V, E)$ is added and updated by *Admin*. This data is used by the *Decentralized router*, as discussed later in Section III.5.

III.3 Network setup

In this section, we discuss the workflow for network setup. Contrary to centralized architecture discussed in Section II.3, here the network is given by *Admin* is first partitioned into regular grids and then stored in *Central Server*. Each RSU then request network for a subset of the grids and store in locally, which is then used by the *Decentralized router*, as discussed later in Section III.5.

III.3.1 Network setup for Central Server

The network setup process for *Central Server* proceeds as follows:

1. The process starts with initial addition of network by the *Admin*. *Admin* sends a graph $G = (V, E)$ and a *geohash precision* value to the *Network Partitioner* in *Central*

Algorithm 1: Partition Network

```
    Data: A graph $G = (V, E)$, $g = \{g_1, g_2, \cdots, g_n\}$, $prec$ = geohash precision value.
1   begin
2       foreach $v \in V$ do
3           $v.grid = gh.encode(v, prec)$;
4       foreach $e_i \in E$ do
5           $(u, v) = e_i$;
6           if $u.grid \neq v.grid$ then
7               Initialize $V_b$ = Set of boundary vertices;
8               $[g_1, g_2, \cdots, g_k] = intersection(e_i, g)$;
9               foreach $g_j \in [g_1, g_2, \cdots, g_k]$ do
10                  $v_i^{j.j+1} = intersection(e_i, g_j, g_{j+1})$;
11                  $v_i^{j.j+1}.grid = (g_j, g_{j+1})$;
12                  $V_b.append(v_i^{j.j+1})$;
13              $[e_1, e_2, \cdots, e_{len(V_b)+1}] = split(e_i, V_b)$;
14              $G.remove(e_i)$;
15              $G.add(V_b)$;
16              $G.add([e_1, e_2, \cdots, e_{len(V_b)+1}])$;
17      foreach $(u, v) \in E$ do
18          $(u, v) = e$;
19          $e.grid = u.grid$;
```

Server. Here *geohash precision* value, as discussed in Appendix A.2, determines the resolution or area of the desired grids.

2. The *Network Partitioner* receives the graph $G = (V, E)$ and *geohash precision* value from the *Admin* and partitions the network into regular grids $\{g_1, g_2, \cdots, g_k\}$ using Algorithm 1. The algorithm proceeds by first annotating each vertex in the network with the grid they belong to, by using *geohash encoding* as discussed in Appendix A.2. Then for each edge in the network, it checks if both the endpoints of the edge are in the same or different grid. If they are in a different grid, then boundary vertices are found by geospatial intersection of a pair of grids and an edge. Then the edges are split by their boundary vertices and the graph is updated accordingly. Finally, all the edges are also annotated with the grid information.

3. As a result of the partition algorithm, each vertex and edge in the graph $G = (V, E)$ is annotated with their respective grid information. This will help in filtering out the graphs for any given grid, thus dividing graph G into subgraphs $\{G^1, G^2, \cdots, G^k\}$, where i in G^i refers to the grid g_i. This partitioned network is then stored in the *Central Server*. Finally, the result of data storage is returned to the *Admin*.

Algorithm 2: Grid Network Mapping

```
Data: A graph $G = (V, E)$, $g = \{g_1, g_2, \cdots, g_k\}$
Result: $\{G^1, G^2, \cdots, G^k\}$
1  begin
2      Initialize GridNetworkList;
3      foreach $g_i \in g$ do
4          Initialize graph $G^i$;
5          foreach $v \in V$ do
6              if $v.grid == g_i$ then
7                  $G^i.add(v)$;
8          foreach $e \in E$ do
9              if $e.grid == g_i$ then
10                 $G^i.add(e)$;
11         GridNetworkList.append($G^i$);
12     return GridNetworkList
```

III.3.2 Network setup for RSU

The network setup process for RSU proceeds as follows:

1. The process starts with each RSU sending a set of grids $\{g_1, g_2, \cdots, g_k\}$ (that has been pre-configured to each RSU), to the *Central Server*.

2. The *Central Server* receives the set of grids $\{g_1, g_2, \cdots, g_k\}$ and maps the network for each grid using Algorithm 2. The partitioned graph stored in *Central Server* already has each vertex and edge in the graph $G = (V, E)$, annotated with their respective grid information. This algorithm is simple and just filters out the graphs for any given grid. So, the response is a set of subgraphs $\{G^1, G^2, \cdots, G^k\}$ where each vertices and edges in a graph G^i maps to grid g_i.

III.4 Training prediction models

We need prediction models to aid our decentralized route planner. Before we discuss in detail each prediction model that is being used in our architecture, we would need to discuss the procedure used to train these data-driven models. This is due to the assumptions placed by us that the data only resides in *RSUs* and not *Central Server*. Hence the standard approach to train data-driven models needs to be changed. Instead of sending data to the *Central Server* for training model, we bring the model training to *RSUs*. We achieve this by leveraging recent advances in Federated Learning (8) which enables *RSUs* to collaboratively learn a shared prediction model while keeping all the training data on the device, decoupling the ability to do machine learning from the need to store the data in the *Central Server*. We briefly describe the Federated learning below.

Federated learning: The goal of Federated Learning is to learn a model with parameters embodied in a real matrix $\mathbf{W} \in \mathbb{R}^{d_1 \times d_2}$, from data stored across all the RSUs. Here $\mathbf{W}$ is a 2D matrix to represent the parameters of each layer in a fully-connected feed-forward network (22). d_1 and d_2 represents the output and input dimensions respectively. the tasks proceed in rounds and each round alternates between local and global model update.

1. **Distribute Global model**: *Admin* randomly initializes the weights $\mathbf{W_0}$ of the prediction model and stores it in *Central Server*. In round $t \geq 0$, the *Central Server* distributes the current model $\mathbf{W}_t$ to a subset S_t of n_t RSUs.

2. **Local update**: Each *RSU* then independently update the model based on their local data. Let the updated local models be $\mathbf{W_t^1}, \mathbf{W_t^2}, \cdots, \mathbf{W_t^{n_t}}$, so the update of each RSU i can be written as $\mathbf{H_t^i} := \mathbf{W_t^i} - \mathbf{W_t}$, for $i \in S_t$. These updates could be a single gradient computed on the RSU, but typically will be the result of a more complex calculation, for example, multiple steps of stochastic gradient descent (SGD) (23) taken on the RSU's local dataset. Each selected RSU then sends the update back to the *Central Server*.

3. **Global update**: It is computed by aggregating all the local updates received from RSUs:

$$\mathbf{H_t} := \frac{1}{n_t}\sum_{i\in S_t}\mathbf{H_t^i}, \mathbf{W_{t+1}} := \mathbf{W_t} + \eta_t\mathbf{H_t}.$$

   Here η_t is learning rate. For simplicity we can choose $\eta_t = 1$.

We use *Federated Learning* procedure to train the prediction models discussed next.

III.4.1 Travel time predictor

To estimate a route with minimum travel time, the search procedure needs to better estimate the time it will take in future to arrive at some edge e_i during exploration. Hence we need to learn travel time predictor $\hat{T}(e_i, \tau_i)$ that estimate travel time on an edge e_i in time interval τ_i. Let edge e_i defines a directed edge from v_i to v_{i+1}, then travel time function can also be defined as $T(v_i, v_{i+1}, \tau_i)$ and here τ_i refers to the departure time at vertex v_i.

For this model, we build a feature set with quantities described in Table III.1. The resulting feature space has 228 dimensions. Since time value τ is categorical, we need to convert it into a form that could be provided to machine learning algorithms to do a better job in prediction. Hence, we used *one-hot encoding* as discussed in Appendix A.1, to map τ_i to one-hot encoded binary for *Week of year, Day of week, Hour of day, Minutes of hour* features.

v_i and v_{i+1} are location data which are represented by geographical coordinates in terms of latitude, longitude pair and are also categorical, but encoding them using *one-hot encoding* can lead to billions of dimensions which can make training inefficient. So, we used *geohash encoding*, as discussed in Appendix A.2 to map v_i and v_{i+1} to geohash encoded binary features *From location, To location* respectively. *geohash encoding* gives control over precision or resolution of the grid represented by each geohash. For vertices in our network, we needed a resolution such that each vertex covers the width of the road. A res-

Table III.1: Feature description for Travel time predictor.

Feature	Dim	Description
From location	42	Geohash encoded binary indication of From coordinate of an edge
To location	42	Geohash encoded binary indication of To coordinate of an edge
Week of year	52	One-hot encoded binary indication of Week of year used to sample travel time data
Day of week	7	One-hot encoded binary indication of Day of week used to sample travel time data
Hour of day	24	One-hot encoded binary indication of Hour of day used to sample travel time data
Minutes of hour	60	One-hot encoded binary indication of Minutes of hour used to sample travel time data
Travel time	1	One-hot encoded binary indication of the true travel time data collected from HERE API.

olution higher than that could cause multiple vertices to belong to the same geohash which can make the model inaccurate, while a resolution lower than that would increase the dimensions of the feature space, take more compute resources and is not necessary. After extensive testing, we found a resolution of 9.5m which uses 42 bits, matches the width of most of the major road segments in the United States.

III.4.2 Equivalent Grid Routing predictor

The search procedure can extend to multiple RSUs. As more RSU are included in the search, it can incur huge delays due to communication costs. Hence, the goal is to minimize the number of message exchanges needed during search and this can happen if we have a good estimate of sequence of grids, a search algorithm go through to reach a destination. Hence we need to learn a Equivalent Grid Routing predictor $\hat{E}$, such that $\hat{E}(s,d,\tau_s)$ gives next best possible grid to reach from source s to travel to destination d and τ_s is the departure time from s.

For this model, we build a feature set with quantities described in Table III.2. The resulting feature space has 255 dimensions. This is similar to the feature description table discussed previously for travel time predictor $\hat{T}$. The only difference here is in the output label which predicts a *geohash* encoded binary of a grid instead. Similarly, here s and

Table III.2: Feature description for Equivalent Grid Routing predictor.

Feature	**Dim**	**Description**
From location	42	Geohash encoded binary indication of From coordinate of an edge
To location	42	Geohash encoded binary indication of To coordinate of an edge
Week of year	52	One-hot encoded binary indication of Week of year used to sample travel time data
Day of week	7	One-hot encoded binary indication of Day of week used to sample travel time data
Hour of day	24	One-hot encoded binary indication of Hour of day used to sample travel time data
Minutes of hour	60	One-hot encoded binary indication of Minutes of hour used to sample travel time data
Next Grid	28	Geohash encoded binary indication of a Grid

d are the location data and are mapped to 42-bit *geohash* encoded binary features *From location, To location* respectively, which has a resolution of 9.5m. Time value τ_s is mapped to one-hot encoded binary for *Week of year, Day of week, Hour of day, Minutes of hour* features.

III.5 Decentralized Route Planning

III.5.1 Algorithm

The goal of the decentralized route planning is to distribute the query among different RSUs. One of the problems we discussed earlier that state of the art solutions for parallelizing the query fails in a time-dependent network. We mitigate this problem by using *Travel time predictor* $\hat{T}$. To minimize the number of RSUs that are communicated during the search or to guided our search efficiently we use *Equivalent Grid Routing predictor* $\hat{E}$. Algorithm 3 handles decentralized routing queries from vehicles. We first list some utility functions that are used in the algorithm and then discuss the algorithm in detail.

Here is a list of utility functions that we have used in our algorithm:

1. $\underline{gh.encode(v)}$: This function uses *geohash encoding* discussed in Appendix A.2 to find the grid to which the vertex belong to.

2. $\underline{\text{GetRSU}(g_i)}$: This function finds the RSU mapping for any grid i

3. $\underline{SP(G,s,d,\tau_s)}$: This function uses network G, Dijkstra algorithm (11) and Travel time predictor $\hat{T}$ to find route from source s at departure time τ_s to destination d with minimum travel time.

4. $\underline{msg(type,RSU_i,val)}$: This is an *async* call for sending the type of message (*type*) and actual message (*val*) to a RSU i. The actual sending of the message to RSU i is handled by a network client. Each of the messages requires different processing and thus the actual message (*val*) they need. In our approach, three types of messages are being handled. They are:

    - **query**: The RSU upon receiving a message of this type, executes Algorithm 3 and pass the actual message (*val*) as arguments to the function that implements this algorithm.
    - **partial path**: The RSU upon receiving a message of this type, executes Algorithm 4 and pass the actual message (*val*) as arguments to the function that implements this algorithm. If the final route plan is returned as a response, it is communicated to the client which made the routing query request.

III.5.2 Example

Figure III.2 shows an example where a network is partitioned into 4 RSUs. For simplicity, this example does not have any redundancy, i.e if a grid g_i is mapped only to one RSU. Figure III.3 shows sequence diagram for this example. The route planning process proceeds as follows:

1. The process starts with a route query (id,s,d,τ_s) from a client or vehicle, where *id* is the unique identifier given by the client identify to this query, s is the source vertex, d is the destination vertex, τ_s is the departure time from s. The route planning process

Algorithm 3: Handle Query

```
Data: A graph $G = (V, E), s \in V, d \in V$, $\tau_s$ = departure time from vertex $s$, $RSU_o$ =
      RSU from where the query origins, $id$: Unique identifier to identify this query.
1  begin
2      $save(id, (s, d, \tau_s))$;
3      $g_s = gh.encode(s)$;
4      $g_d = gh.encode(d)$;
5      if $g_s \neq g_d$ then
6          $g_{\text{next}} = \hat{E}(s, d, \tau_s)$;
7          $RSU_{\text{next}}$ = GetRSU$(g_{\text{next}})$;
8          $\{v_1, v_2, \cdots, v_b\} = g_s.intersect(g_d)$;
9          foreach $v \in \{v_1, v_2, \cdots, v_b\}$ do
10             $\hat{t}^s_v = \hat{T}(s, v, \tau_s)$;
11             $msg$(“query”,$RSU_{\text{next}}, \{id, v, d, \tau_s + \hat{t}^s_v, RSU_o\}$);
12         foreach $v \in \{v_1, v_2, \cdots, v_b\}$ do
13             $R_p = SP(G, s, v, \tau_s)$;
14             $msg$(“partial path”,$RSU_o, \{id, s, v, \tau_s, R^s_v\}$);
15     else
16         $R_p = SP(G, s, d, \tau_s)$;
17         $msg$(“partial path”,$RSU_o, \{id, s, d, \tau_s, R^s_d\}$);
```

is decentralized, asynchronous and each RSU might be processing multiple requests simultaneously, hence we needed a *id* to differentiate each request. Please note that a query can be sent to any RSU. For simplicity, let us assume that the client sends this route query to the nearest RSU, which is RSU_1.

2. Processing at RSU_1: Here are the sequence of steps executed in RSU_1:

   (a) It receives the route query (id, s, d, τ_s) from client and calls Algorithm 3 to find route. Since the source s and destination d do not belong to the same grid, then *Equivalent Grid predictor* is used find the next best possible grid to reach destination d. Let us assume that the grid is g_2, which is present in RSU_2.

   (b) Then the algorithm finds the nodes at the intersection of the current grid (grid which has s and the next best grid. We call these nodes as *boundary nodes*. In our example, there is only one boundary node v_{12}.

Algorithm 4: Handle Partial Path

```
Data: A graph $G = (V,E), u \in V, v \in V$, $\tau_u$ = departure time from vertex $u$, $R$ =
      Route from $u$ to $v$, starting at $\tau_u$, $id$: Unique identifier to identify this query.
Result: Final Route plan $(id, R_{final})$ or NULL
1  begin
2      $(s,d,\tau_s) = get(id)$;
3      if $v == d$ then
4          $R_{final} = SP(G_{id}, s, d, \tau_s)$;
5          return $(id, R_{final})$;
6      else
7          $G_{id}$ = GetGraph(id);
8          if $G_{id} == NULL$ then
9              Initialize Graph $G_{id}$;
10         foreach $(e_i, \tau_i) \in R$ do
11             $G_{id}.add(e_i)$;
12             $G_{id}[e_i] = (\tau_i, \tau_{i+1} - \tau_i)$;
13         SaveGraph(id, $G_{id}$);
14         return $NULL$;
```

(c) After getting all the boundary vertices, for each boundary vertex, the algorithm uses *Travel time predictor* to estimate the time it will take to reach each the boundary vertex. In our example, we got $\hat{t}^s_{v_{12}}$ as the estimated travel time to reach v_{12} from source s and departure time τ_s from s. So the estimated time of arrival at v_{12} from s is $\tau_s + \hat{t}^s_{v_{12}}$ and we denote it by $\hat{\tau}_{v_{12}}$.

(d) Then an asynchronous message $(\text{``}query\text{''}, id, v_{12}, d, \hat{\tau}_{v_{12}}, RSU_1)$ is sent to RSU_2.

(e) After sending the message, the RSU_1 proceeds in finding actual route from s to boundary vertex v_{12}. Let R^s_{12} be the route, indicating that the route is from vertex s to v_{12}. After getting the route R^s_{12}, a message $(\text{``}partialpath\text{''}, id, s, v_{12}, \tau_s, R^s_{12})$ is prepared. This message is meant to be sent to the RSU to which the client sent the request. Since it's RSU_1, which is itself, a function call is made to handle this message where the function arguments are the same as the message. This function implements Algorithm 4 which handles the partial routes or paths. We call it *partial route* because this is still not the final route that needs to be given

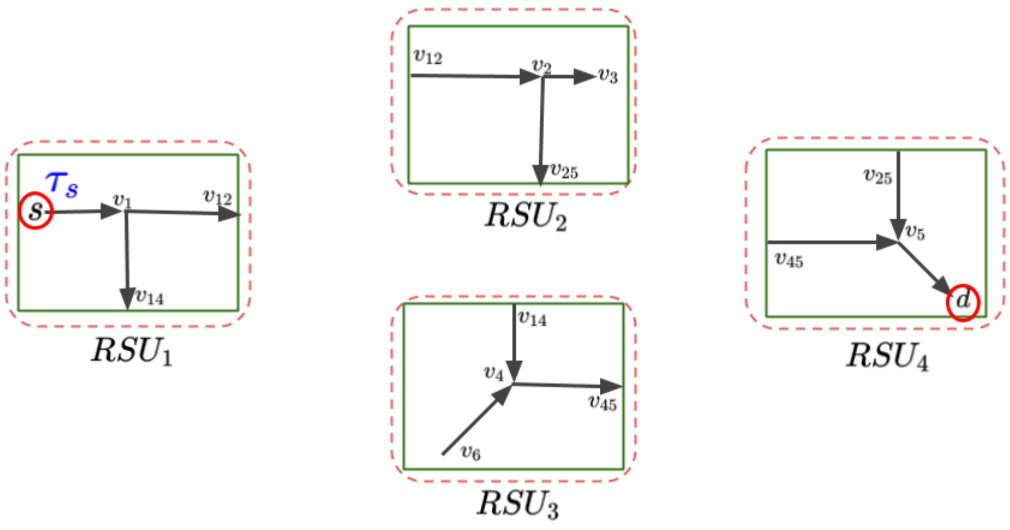

Figure III.2: Decentralized Route Planning example

to the client.

(f) The goal of Algorithm 4 is to create a new graph with With the *id* given by the request or get it if it exists already, add all partial routes to the graph and finally, do a simple shortest path routing on it. Since the route R^s_{12} does not have the destination, the algorithm just saves the graph with the *id* given by the request.

(g) At this step, *RSU* waits from partial routes from other RSUs for this request identified by its *id* and executes 4 if it receives a message with the partial route in it until it gets a partial route which has the destination in it.

3. Processing at RSU_2: Here are the sequence of steps executed in RSU_2:

(a) It receives route query $(id, v_{12}, d, \hat{\tau}_{v_{12}}, RSU_1)$ from RSU_1 and calls Algorithm 3 to find route. Since the source v_{12} and destination d do not belong to the same grid, then *Equivalent Grid predictor* is used find the next best possible grid to reach destination d. Let us assume that the grid is g_4, which is present in RSU_4.

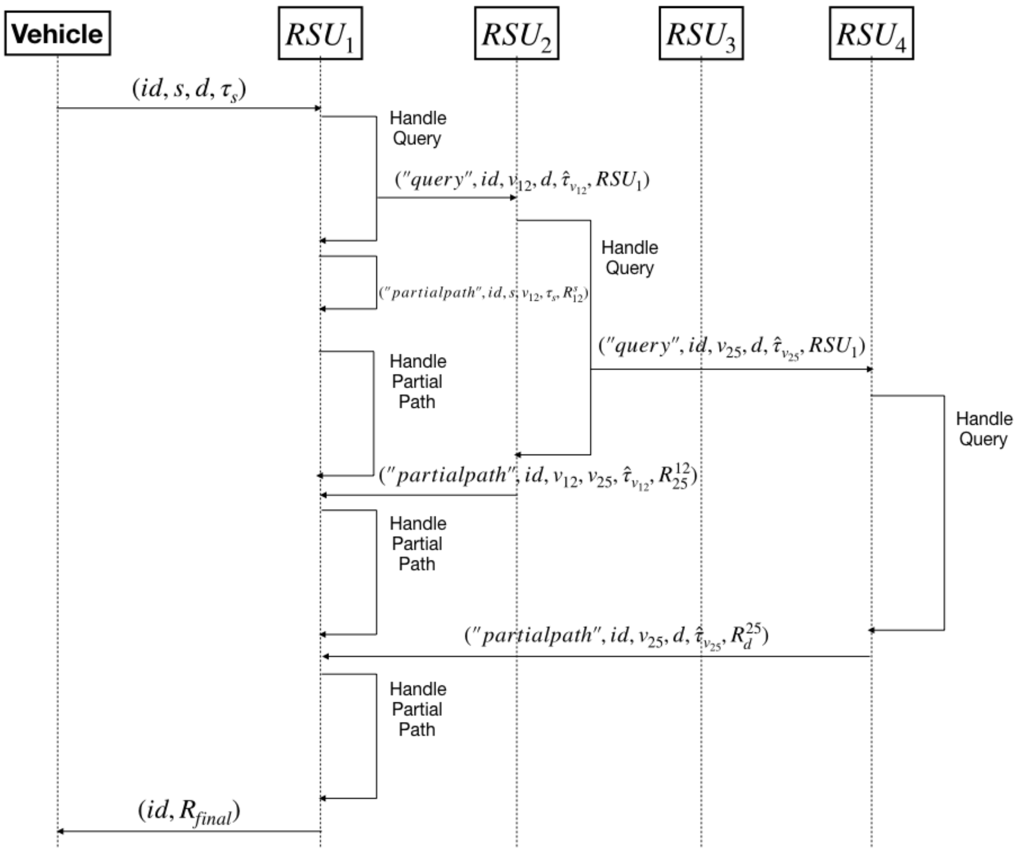
Vehicle
RSU_1
RSU_2
RSU_3
RSU_4
(id, s, d, τ_s)
Handle Query
$("query", id, v_{12}, d, \hat{\tau}_{v_{12}}, RSU_1)$
Handle Query
$("partialpath", id, s, v_{12}, \tau_s, R_{12}^s)$
$("query", id, v_{25}, d, \hat{\tau}_{v_{25}}, RSU_1)$
Handle Partial Path
Handle Query
$("partialpath", id, v_{12}, v_{25}, \hat{\tau}_{v_{12}}, R_{25}^{12})$
Handle Partial Path
$("partialpath", id, v_{25}, d, \hat{\tau}_{v_{25}}, R_d^{25})$
Handle Partial Path
(id, R_{final})

Figure III.3: Sequence Diagram of Decentralized Route Planning example

(b) Following similar steps in Algorithm 3 as discussed earlier, we find v_{25} be the boundary vertex. An asynchronous message $(\text{“}query\text{”}, id, v_{25}, d, \hat{\tau}_{v_{25}}, RSU_1)$ is sent to RSU_4.

(c) After sending the message, the RSU_2 proceeds in finding actual route from v_{12} to boundary vertex v_{25}, with departure time $\hat{\tau}_{v_{12}}$ from v_{12}. It then sends this partial route to RSU_1, which is RSU from which the route query originated.

4. Processing at RSU_4: Here are the sequence of steps executed in RSU_4:

(a) It receives route query $(id, v_{25}, d, \hat{\tau}_{v_{25}}, RSU_1)$ from RSU_2 and calls Algorithm 3 to find route. Since the source v_{25} and destination d belong to the same grid, a simple shortest path routing is done to find actual route. Then it sends this partial route to RSU_1, which is RSU from which the route query originated.

5. Finally, RSU_1 gets partial route from RSU_4 which has a partial route to d in it. At this step, Algorithm 4 executes shortest path routing on the graph identified by the request identifier id and the final route plan (id, R_{final}) is sent to the client.

III.5.3 Properties

As discussed in Chapter II, A^* is a classic algorithm for informed search. It relies on a heuristic function, which for any given vertex, gives the estimated cost to reach the destination. This function guides the search procedure. Our approach for route planning is essentially an informed search procedure where *Equivalent Grid Routing Predictor* acts as a heuristic function. So, we compare our approach to A^* and discuss how it fares with them on two important criteria that any informed search procedure should have:

- Termination and Completeness: It is well established that if a graph G is finite and edge weights are negative, then A^* is guaranteed to terminate and is complete, i.e it will always find a route from source to destination if one exists. Our approach is similar to A^*, but it cannot give guarantees on its termination and hence may not be complete. It is because *Equivalent Grid Routing Predictor* used in our approach cannot guarantee that its predictions are always true.

  Furthermore, the decentralized nature of our approach can violate this condition, such as during failures, communication failure, or if RSUs do not have memory and needs to delete the grids they store and so on. This can be mitigated however if all the grids are active, i.e must be present in at minimum in one RSU and it must be reachable, i.e no communication failure. Hence a proactive approach is needed to guarantee this state in the cluster of RSUs.

- Admissibility: It is also well established that A^* guarantees to return an optimal solution if the heuristic function it uses, is admissible, i.e at any given vertex the estimated cost given by the heuristic function must always be lower than or equal to the actual cost of reaching the goal state. Our approach cannot guarantee this too. It is because *Travel time predictor* does not guarantee its travel time predictions and similarly, *Equivalent Grid Routing Predictor* cannot guarantee that the next possible grid it predicts is always true.

III.5.4 Metrics for evaluating performance

To measure the performance of our decentralized search algorithm, we use the following metrics:

- Space complexity: We discussed in the previous section that our algorithm uses Dijkstra to find partial routes. Dijkstra algorithm has a space complexity of $O(|V|^2)$, where $|V|$ is the number of vertices in graph G. Since our graph is divided among RSUs, space complexity should be less than $O(|V|^2)$ on average and it depends on the network and mapping of grids to RSUs. If all grids are mapped to same RSU, then $O(|V|^2)$ is the space complexity.

- Time complexity: The time complexity of Dijkstra is $O(|E|log|V|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in graph G. Similar to our previous discussion on *Space complexity*, since our graph is divided among RSUs, the time complexity should also be less than $O(|E|log|V|)$ in each RSU. However this might not hold for following reasons: (1) multiple RSUs are finding partial routes simultaneously, (2) the time complexity of our approach depends on the accuracy of *Equivalent Grid Routing Predictor*, (3) the communication overhead among RSUs depends on the accuracy of *Equivalent Grid Routing Predictor*. Hence, we cannot give theoretical bounds on the time complexity of our approach.

- Route inefficiency: Even if the search completes successfully, the routes given by our approach or in general any search procedure that uses a data-driven travel time prediction model, is not guaranteed to be optimal. If R^* is the optimal route given by a search algorithm by using actual location collected and $\hat{R}$ is the route given by our approach, then $|len(R^*) - len(\hat{R})|$ tells how far our route is from optimal or how inefficient our route is. If $len(R^*) - len(\hat{R})$ is negative, then our approach overestimated the travel time and if $len(R^*) - len(\hat{R})$ is positive, then our approach underestimated the travel time.

CHAPTER IV

Experiments and Results

In this chapter, we evaluate our decentralized architecture for route planning. We needed a moderately scaled region where we could get required datasets and conduct our experiments. Hence, we chose Nashville metropolitan region for our case study. First, we discuss our experimental setup and the data used in our experiments, then we evaluate different prediction models used in our work and finally, we evaluate our approach and compare it with state of the art solutions for centralized route planning.

IV.1 Evaluation metrics

Before we discuss the set up of our experiment and evaluate our approach on large scale using Nashville metropolitan region, we want to discuss the metrics used in our experiments to evaluate the prediction models in Section IV.3 and our decentralized route planning approach in Section IV.4. We discuss this with an example route query from Institute for Software Integrated Systems to Music City Central, Nashville, TN by car and the departure time for the request is at 9 AM. Figure IV.1 shows the estimated route returned by our approach (denoted by $\hat{R}$) and the optimal route given by Dijkstra shortest path algorithm (denoted by R^*). To evaluate the prediction models we used the following metrics:

- **Error**: *Travel time predictor* estimates travel time, hence it is a regression model and we use Mean Absolute Error (MAE) (24) for this. On the other hand, *Equivalent Grid Routing predictor* estimates best possible grid, hence it is a classification model and we use Cross-Entropy Loss (25) here. If the loss is MAE, then the unit is **minutes**, while Cross-Entropy Loss has not units.

- **Route inefficiency**: The error functions discussed earlier only helps in estimating how good a model predicts compared to a test set of ground truth values. Errors

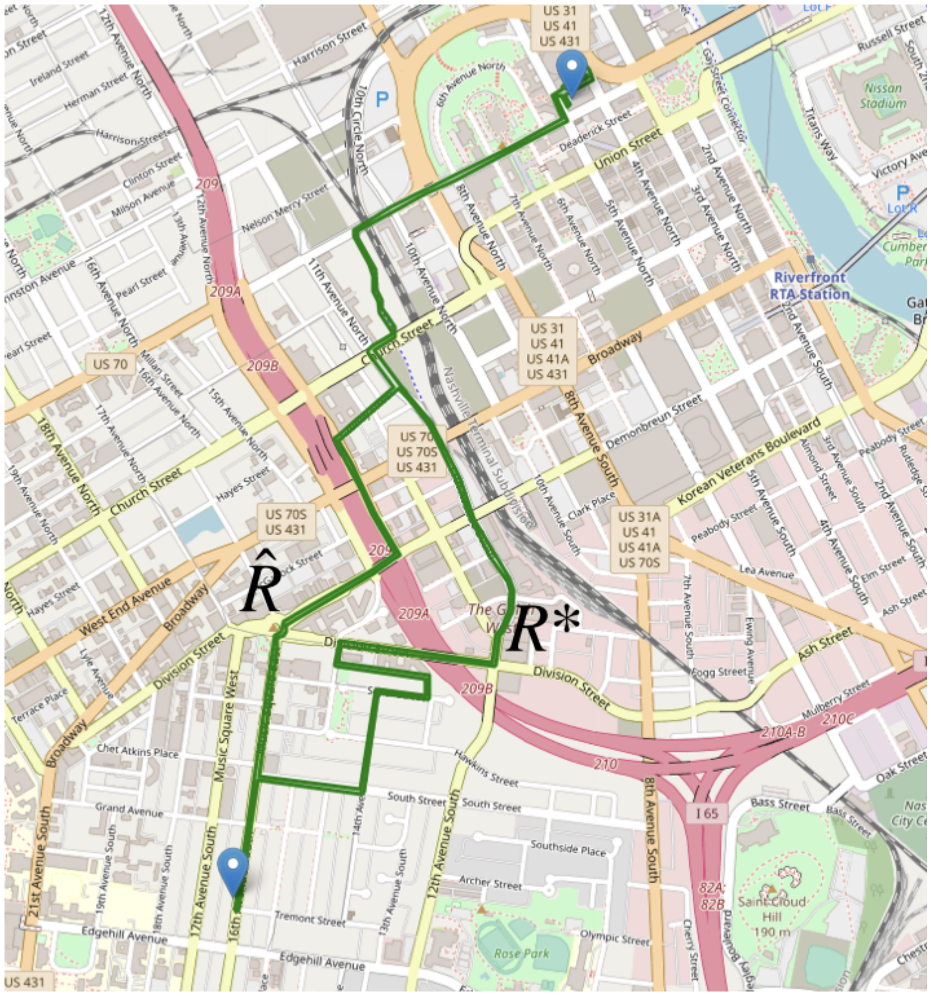

Figure IV.1: Example showing inefficiency between an optimal route R^* and an estimated route $\hat{R}$ returned by our decentralized route planner.

in predictions can lead to error in shortest path evaluation and may result in inefficient routes. We discuss this metric in detail in the previous chapter. If we consider our example, the route $\hat{R}$ given by our approach has a travel time of approximately $len(\hat{R}) = 7mins$ and the optimal route R^* has a travel time of approximately $len(R^*) = 9mins$. Hence, here the inefficiency is $|len(R^*) - len(\hat{R})| = 2mins$ and since $len(R^*) - len(\hat{R})$ is positive, our approach underestimated the travel time.

- **<u>CPU per RSU</u>**: For this metric, we periodically log the change in CPU consumption for each RSU, while the training is being done. We are particularly interested in the median and maximum **change in CPU consumption** for an RSU.

- **Memory per RSU**: Similar to CPU consumption, we periodically log the change in Memory consumption for each RSU, while the training is being done and report the median and maximum **megabytes consumed (MB)** for an RSU.

- **# Messages**: Total number of message sent during training phase only pertains to the Federated learning. In Central learning, the training is done on a single machine. Hence, there is no message passing during Central learning.

To evaluate our decentralized route planning approach we used the following metrics:

- **CPU per RSU**: For this metric, we periodically log the change in CPU consumption for each RSU, while the routing queries are being made. We are particularly interested in the median and maximum **change in CPU consumption** for an RSU.

- **Memory per RSU**: Similar to CPU consumption, we periodically log the change in Memory consumption for each RSU, while the routing queries are being made and report the median and maximum **megabytes consumed (MB)** for an RSU.

- **Query time per trip request**: For this metric, we log the response time in **seconds**, for each route query and get median and maximum travel time in minutes, for a trip. Note that, we are not logging the number of messages here since we are more interested in the query response time for an algorithm rather than the number of the message passed. The query time metric already captures the overhead due to the message passing.

Table IV.1: Regions covered by RSUs.

RSU	Bounding box
RSU_1	(-87.1875, 36.21093, -86.83593, 36.38671)
RSU_2	(-86.83593, 36.21093, -86.48437, 36.38671)
RSU_3	(-87.1875, 36.03515, -86.48437, 36.21093)
RSU_4	(-87.1875, 35.85937, -86.83593, 36.03515)
RSU_5	(-86.8359, 35.85937, -86.48437, 36.03515)

IV.2 Experiment setup

In this section, we discuss some necessary setup needed before we can start evaluating the prediction models in and our decentralized route planning approach. The necessary configurations are listed below:

1. **RSUs**: Cluster of 5 RSUs simulated by Docker[1] containers. RSUs are static, i.e their location does not change with time. Table IV.1 shows exact bounding boxes, which are the regions covered by RSUs in Nashville. Figure IV.2 shows the RSUs and the region covered by RSUs in the map of Nashville.

2. **Network setup for Central Server**: We use OpenStreetMap[2] data for Nashville metropolitan area whose bounding box is $(-87.04999, 35.97, -86.510, 36.42)$. There are a total of $233,123$ nodes and $474,213$ edges in this region.

   For partitioning the network within this region, in *Central Server* using the Algorithm 1 discussed earlier, we used a *geohash precision* value of 28 bits, which gives grids area of $1.44km^2$. A total of 1034 grids are created as a result of partition.

   Our reason for choosing grid area of $1.44km^2$ is *adhoc*. Grid area affects the resource consumption of RSU. One approach is to conduct a lot of tests and get a fair estimate of how much area RSU can contain depending on its memory constraints. Another approach is to have grids with a small area and move grids among RSUs at runtime. We chose the latter and have smaller grids with an area of $1.44km^2$.

[1] https://www.docker.com/
[2] https://www.openstreetmap.org

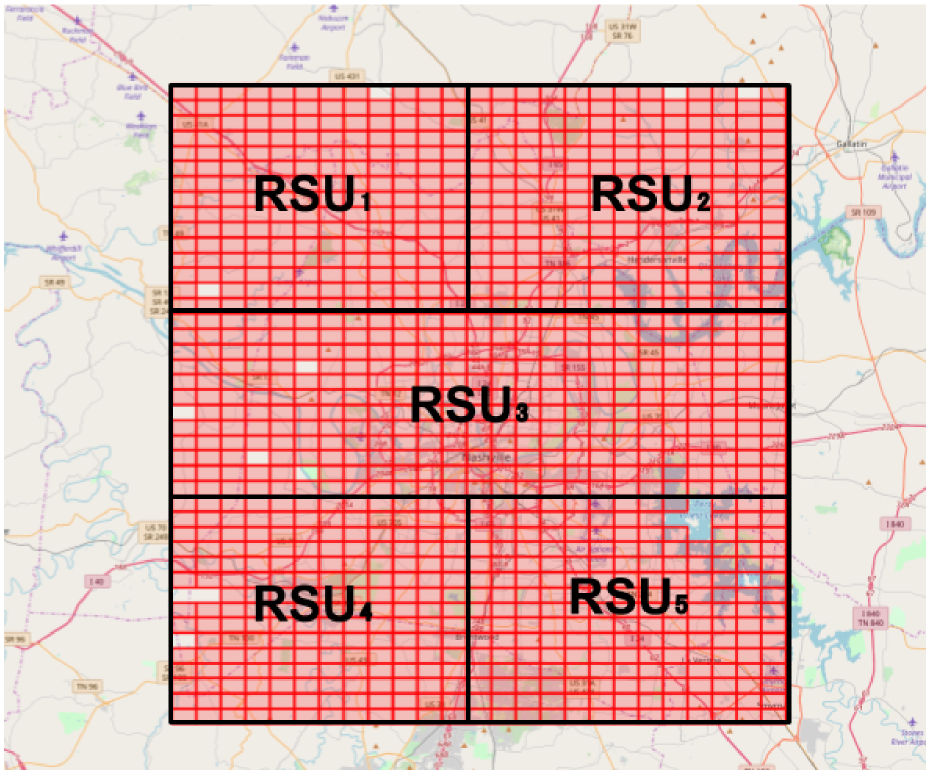

Figure IV.2: Partition of Nashville metropolitan area into grids of area $1.44km^2$ and placement of grids in RSUs

3. **Network setup for RSUs**: Now that the geographical region for each RSU and partition of the region into bounding boxes are done, each RSU can now get graphs for their grids using Algorithm 2. We have assumed that our cluster of 5 RSUs never runs out of memory and hence we do not move grids among RSUs at runtime. Figure IV.2 shows the placement of grids in RSUs.

4. **Location data**: To simulate vehicle locations in the region, we use historical traffic data collected at an interval of 1 minute, via the HERE API[3] for the Nashville metropolitan area. We do not know how HERE API gets actual traffic speed values for road segments and hence we assume that the traffic speed reported by them are accurate. Traffic data from January 1 to January 31, 2018, is used for training and data from Feb 1 to Feb 7, 2018, is used for testing.

[3]https://www.here.com/en

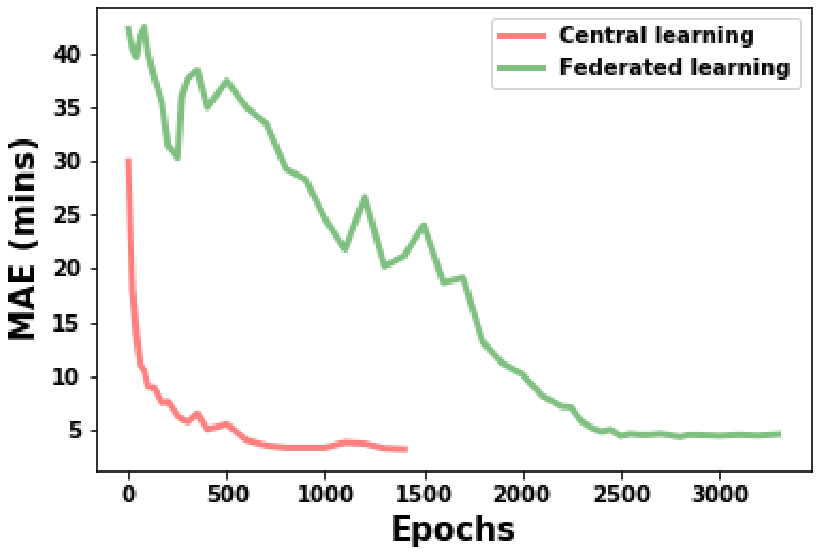

Figure IV.3: MAE vs Epoch curve during training of Travel time predictor

5. **Trip data**: To simulate routing queries from vehicles, we synthetically generate 1,000,000 source and destination pairs are chosen randomly from the Nashville metropolitan area. For each of these source-destination pairs, departure times were chosen uniformly from 9am-5pm. 800,000 trips have departure times from January 1 to January 31, 2018, and 200,000 trips have departure times from Feb 1 to Feb 7, 2018.

IV.3 Evaluation of prediction models

IV.3.1 Travel time predictor

We use a deep feed-forward neural network (DNN) (26) for a regression that estimates travel time of an edge. Extensive tuning both in the configuration of hidden layers and the activation and optimization functions was done during training. *SGD* (23) is chosen as optimizer for the neural network. The configuration for hidden layers were chosen as $[200, 190, 170, 150, 100, 50, 20, 10]$. Early stopping criteria are employed to avoid overfitting. Fig. IV.3 the change in validation Mean Absolute Error (MAE) with epoch steps during the training phase of this predictor.

Table IV.2: Resource consumption for Travel time predictor.

	CPU per RSU (% used)	**Memory per RSU (MB)**	**# Messages**
Central Learning	78% (median) 97% (max)	191 (median) 307 (max)	N/A
Federated Learning	67% (median) 84% (max)	51 (median) 88 (max)	6255

Figure IV.3 shows the Federated learning took more time to train than Central learning. MAE for a model trained with Central learning is 1.16 minutes which is less than the model trained from Federated learning– 3.31 minutes. More importantly, errors in travel time prediction model led to 11.4% of route queries return routes with 1.3 - 1.7 times decreased travel times than the actual.

Table IV.2 evaluates the resource consumption of this model when trained with Federated learning and Central learning. Results show that Federated learning uses less CPU than Central learning in both models. Federated learning uses 3.4 - 3.7 times less Memory per node than Central learning, which was expected since each node use their local model and data while training compared to Central learning which is trained in a single node with all data. Federated learning sent 6255 messages while training, while it is not applicable in Central learning since the training happens locally in a single node.

IV.3.2 Equivalent Grid Routing predictor

We use a deep feed-forward neural network (DNN) (26) for a binary classification that gives the next best possible grid for a given pair of source, destination along with the time interval. For binary classification, the activation function used in the output layer is a sigmoid function (27). Extensive tuning both in the configuration of hidden layers and the activation and optimization functions was done during training. *Adam optimizer* is chosen as an optimizer for the neural network. The configuration for hidden layers were chosen as $[250,200,170,100,50,20,10]$. Fig. IV.4 shows the loss vs epoch curve during the training

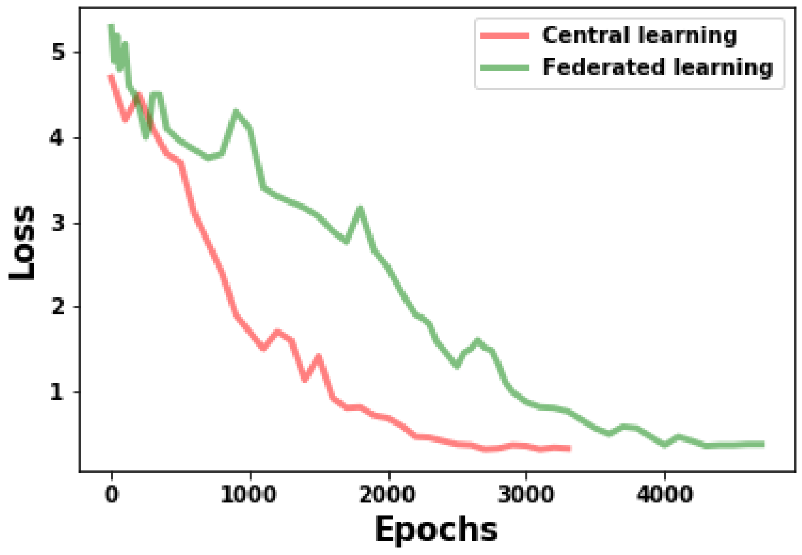

Figure IV.4: Loss vs Epoch curve during training of Equivalent Grid Routing predictor

phase for this predictor. Federated learning took more time to train than Central learning. The loss for a model trained with Central learning is 0.32 which is less than the model trained from Federated learning– 0.37. More importantly, errors in Grid prediction model led to no route for 0.8% of users and 7.6% of routing queries returned inefficient routes, i.e routes with increased or decreased travel time than actual.

Table IV.3 evaluates the resource consumption of this model when trained with Federated learning and Central learning. Results show that Federated learning uses less CPU than Central learning on average, while Federated learning used more CPU on max. There is no clear reason why. Federated learning uses 3.3 - 3.6 times less Memory per node than Central learning, which was expected since each node use their local model and data while training compared to Central learning which is trained in a single node with all data. Federated learning sent 9543 messages while training, while it is not applicable in Central learning since the training happens locally in a single node.

Table IV.3: Resource consumption for Equivalent Grid Routing predictor.

	CPU per RSU (% used)	**Memory per RSU (MB)**	**# Messages**
Central Learning	81% (median) 93% (max)	217 (median) 336 (max)	N/A
Federated Learning	74% (median) 97% (max)	64 (median) 91 (max)	9543

Table IV.4: Evaluation of routing algorithms.

Algorithm	**CPU per RSU (% used)**	**Memory per RSU (MB)**	**Query time per trip request (s)**
Single server Dijkstra	23% (median) 31% (max)	5.78	0.97 (median)
Parallel Dijkstra	27% (median) 36% (max)	0.76 (median) 1.14 (max)	9.2 (median) 19.13 (max)
Contraction Hierarchies	18% (median) 23% (max)	13.36	0.016 (median)
Parallel Contraction Hierarchies	13% (median) 21% (max)	3.31 (median) 5.79 (max)	5.78 (median) 10.21 (max)
Our approach	52% (median) 67% (max)	0.94 (median) 1.31 (max)	2.43 (median) 5.81 (max)

IV.4 Evaluation of Decentralized Route planner

For evaluating our route planning approach, we use the metrics discussed in Section III.5.4. For evaluating *Space complexity* we measure memory consumption and similarly for evaluating *Time complexity* we measure *Query response times*. In addition to that, we also measure *CPU consumption* to evaluate parallelism of our approach.

Table IV.3 evaluates the resource consumption of our approach and compares it with state of the art solutions for centralized route planning in a time-independent. We summarize the results below:

CPU consumption:

- Our approach uses more CPU than Single server Dijkstra or Parallel Dijkstra because in our approach shortest paths are calculated between boundary nodes in parallel when the request is received.

- Contraction Hierarchies and Parallel Contraction Hierarchies uses less CPU than rest. Its because they consume more memory for caching shortcut edges, which pays off during query time when there are very few edges to explore.

Memory consumption:

- Our approach uses less Memory than Single server Dijkstra because the network is divided among different nodes contrary to single server Dijkstra.
- Our approach uses more Memory than Parallel Dijkstra, because in our approach inference models are used, which takes up more runtime memory contrary to Parallel Dijkstra.
- Contraction Hierarchies and Parallel Contraction Hierarchies uses far more Memory than rest. Its because they consume more memory for caching shortcut edges, which pays off during query time when there are much fewer edges to explore.

Query response times:

- Single server Dijkstra and Contraction hierarchies has far less query time than our approach because there is no network communication involved.
- Contraction hierarchies have the least query time.
- Parallel algorithms such as parallel Dijkstra and parallel Contraction hierarchies have higher query times than our approach since their search proceeds sequentially because of time-dependency of the network.
- Parallel Dijkstra has the highest query times because there are no shortcut edges cached such as in parallel Contraction hierarchies and thus the search space is not guided. Our Grid prediction model guides our search, thus incur fewer query times.

CHAPTER V

Conclusion and Future work

Route planning in a time-dependent network is relevant to many problems faced by cities. Centralized approach for planning routes is a well-studied topic and there are a lot of studies done on optimizing the algorithm to decrease response times for any query from the client. Some techniques have a pre-processing phase whose goal is to reduce the search space during query time. Other techniques use a shared memory model to plan routes in a parallel manner. The central approach for route planning does reduce network latency and also makes it efficient since all the data management can be done centrally and can be horizontally scaled if more resources are needed.

While there are many advantages to central route planning, they fell short in solving the problem that motivated this thesis, which is to help cities with a limited budget and network coverage, provide self-sustaining mobility services for its residents while still preserving their privacy. Our goal was to use the limited compute resources available to the city and are dispersed throughout the region, to plan routes in a time-dependent network. To this end, we presented a decentralized approach for route planning in a time-dependent network, where the computing devices have limited resources. We leveraged recent advances in federated learning to collaboratively learn shared prediction models online while keeping all the training data on the device, thus preserving privacy. Experiments show that our approach uses 1.2 - 6 times less memory per compute node than central approach and has moderately higher query response time than central server approach, but our decentralized architecture makes it resilient in case of system failures and has high scalability.

The core of our architecture relies on data-driven models that estimates travel times and guides the search procedure during query time. However, there are still some problems that should be further investigated before it can be deployed in production. First, further studies are needed to improve travel time and grid prediction models to mitigate the impact of errors on user trips. Second, our approach does not guarantee termination and completion of algorithms, hence further studies are needed to develop data-driven models that can enable us to get theoretical bounds for our search procedure. Third, more experiments are needed to test our architecture for a different system and network failures and evaluate the resiliency of our approach. Finally, our approach can be extended to allow multiple modes of transportation and integrate with a decentralized computation market platform.

Appendix A

Encoding

A.1 One-hot encoding

Categorical problems are quite commonplace in machine learning problems and are more challenging to deal with. In particular, many machine learning algorithms require that their input is numerical to perform gradient descent properly and there is not necessarily any ordering between categories (e.g. a feature ’CompanyName’ with names of companies such as ’Honda’, ’Volkswagen’, ’Boeing’ and so on.). Hence, categorical features must be transformed into numerical features before we can use any of these algorithms. In One-hot encoding, for each unique value in a feature (say ’Honda’), one column is created (say ”CompanyName-Honda”), where the value is 1 if for that instance the original feature takes that value and 0 otherwise. So, the number of dimensions that are added, is equal to the number of unique values in the categorical feature.

A.2 Geohash encoding

Geographical locations are also considered as a categorical feature since there is no ordering among geographical locations. Using One-hot encoding to represent geographical location data can lead to billions of dimensions and the training becomes difficult. Geohash (28) converts geographical coordinates to a short alphanumeric or binary string. Greater *precision* allows greater precision or resolution. For example, with 28 bits, Geohash encoding can encode a geographical coordinate with a resolution of 2.8km. With 42 bits, Geohash encoding can encode a geographical coordinate with a resolution of 9.5m.

BIBLIOGRAPHY

[1] United Nations Department of Economic and Social Affairs, "68% of the world population projected to live in urban areas by 2050," 2018, [Online; accessed 22-July-2019]. [Online]. Available: https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html

[2] A. Mukhopadhyay, G. Pettet, C. Samal, A. Dubey, and Y. Vorobeychik, "An online decision-theoretic pipeline for responder dispatch," in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ACM, 2019, pp. 185–196.

[3] Omnigo, "911 dispatch software," 2019. [Online]. Available: https://www.omnigo.com/solutions/computer-aided-dispatch-software

[4] P. Sanders and D. Schultes, "Engineering highway hierarchies," in *ESA*, vol. 6. Springer, 2006, pp. 804–816.

[5] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," *Experimental Algorithms*, pp. 319–333, 2008.

[6] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[7] Businness Insider, "Morgan stanley: 75 billion devices will be connected to the internet of things by 2020," https://www.businessinsider.com/75-billion-devices-will-be-connected-to-the-internet-by-2020-2013-10?IR=T, 2013.

[8] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[9] T. Pajor, "Multi-modal route planning," *Universität Karlsruhe*, 2009.

[10] T. A. Manual, "Bureau of public roads," *US Department of Commerce*, 1964.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[12] L. R. Ford Jr, "Network flow theory," RAND CORP SANTA MONICA CA, Tech. Rep., 1956.

[13] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[14] G. Dantzig, *Linear programming and extensions*. Princeton university press, 2016.

[15] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[16] A. V. Goldberg and R. F. F. Werneck, "Computing point-to-point shortest paths from external memory." in *ALENEX/ANALCO*, 2005, pp. 26–40.

[17] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *European Symposium on Algorithms*. Springer, 2005, pp. 568–579.

[18] G. Di Stefano, A. Petricola, and C. Zaroliagis, "On the implementation of parallel shortest path algorithms on a supercomputer," in *International Symposium on Parallel and Distributed Processing and Applications*. Springer, 2006, pp. 406–417.

[19] Y. Tang, Y. Zhang, and H. Chen, "A parallel shortest path algorithm based on graph-partitioning and iterative correcting," in *2008 10th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2008, pp. 155–161.

[20] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1998, pp. 722–731.

[21] C. Vetter, "Parallel time-dependent contraction hierarchies," *Student Research Project*, p. 134, 2009.

[22] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[23] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[24] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.

[25] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Advances in neural information processing systems*, 2018, pp. 8778–8788.

[26] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT Press, 2016, vol. 1.

[27] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[28] T. Vukovic, "Hilbert-geohash-hashing geographical point data using the hilbert space-filling curve," Master's thesis, NTNU, 2016.