Partition WaveNet for Deep Modeling of Automated Material Handling System Traffic

by

David J. Amirault

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2020

Certified by. . . . .
Duane Boning
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Partition WaveNet for Deep Modeling of Automated Material Handling System Traffic

by

David J. Amirault

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The throughput of a modern semiconductor fabrication plant depends greatly on the performance of its automated material handling system. Spatiotemporal modeling of the dynamics of a material handling system can lead to a multi-purpose model capable of generalizing to many tasks, including dynamic route optimization, traffic prediction, and anomaly detection. Graph-based deep learning methods have enjoyed considerable success in other traffic modeling domains, but semiconductor fabrication plants are out of reach because of their prohibitively large transport graphs. In this thesis, we consider a novel neural network architecture, Partition WaveNet, for spatiotemporal modeling on large graphs. Partition WaveNet uses a learned graph partition as an encoder to reduce the input size combined with a WaveNet-based stacked dilated 1D convolution component. The adjacency structure from the original graph is propagated to the induced partition graph. For our problem, we determine that supervised learning is preferable to reinforcement learning because of its flexibility and robustness to reward hacking. Within supervised learning, Partition WaveNet is superior because it is both end-to-end and incorporates the known spatial information encoded in the adjacency matrix. We find that Partition WaveNet outperforms other spatiotemporal networks using network embeddings or graph partitions for dimensionality reduction.

Thesis Supervisor: Duane Boning
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank Samsung, and in particular Mokmin Park (`mmpark@mit.edu`), for their partnership on this project. Without their collaboration, this research and our experiments using the proprietary industry Fab emulator would not have been possible. I would like to thank Professor Duane Boning for being an incredible thesis supervisor and mentor. His tireless efforts, breadth of knowledge about manufacturing systems, and insightful feedback helped shape this project into what it is today. I would like to thank Jami Mitchell for her work organizing the Boning group and coordinating our working and meeting spaces. I would like to thank the members of the Boning group for their research advice and helpful commentary during our weekly group meetings. I would like to thank my friends and family, and especially my parents, for their encouragement and affection throughout my academic endeavors. Last but not least, I would like to thank Carmen Chan for her love and support.

Contents

1 Introduction **13**

1.1 The Task of an AMHS 14

1.2 Related Works 15

1.2.1 Spatiotemporal Graph Neural Networks 15

1.2.2 Dimensionality Reduction 16

1.3 Thesis Organization 18

2 Problem Framing **21**

2.1 Reinforcement Learning 21

2.2 Supervised Learning 24

2.3 WaveNet in Practice 27

3 Methods **29**

3.1 Problem Formulation 29

3.2 Partition-Embedded Graph Convolution Layer 30

3.3 Temporal Convolution Layer 32

3.4 Architecture 33

4 Simulation **35**

4.1 Goals 35

4.2 Implementation 36

4.3 Preventing Deadlock 38

4.4 Design Abstractions 42

4.4.1 AppSite 42
4.4.2 RailPath 43
4.4.3 FOUP 43
4.4.4 OHT 44
4.4.5 MHS 44
4.4.6 AMHS 45
4.4.7 Simulation 45
4.4.8 Evaluation 46
4.4.9 Animation 46
4.5 Summary Statistics 47
4.6 Data Generation 49
4.7 Containerization 53

5 Experiments **59**
5.1 Constructed Datasets 59
5.2 Benchmark Models 63
5.2.1 Selected Benchmark Models 63
5.2.2 Rejected Benchmark Models 65
5.3 Computational Setup 66
5.4 Experimental Results 67
5.5 Future Analyses 71
5.6 Further Directions 72

List of Figures

1-1 Vehicle traveling along the OHT system of a Fab 14
1-2 Graph convolution 16
1-3 Graph partition 17
1-4 Network embedding 18

2-1 Toy reinforcement learning problem 22
2-2 Types of machine learning 24

3-1 Stacked dilated 1D causal convolution 33
3-2 Graph WaveNet architecture 34

4-1 OHT system graph layout 37
4-2 Traffic jam vulnerability 41
4-3 Animation snapshot 47

5-1 Hand-selected graph partition 62
5-2 Deep autoencoder architectures 73

List of Tables

4.1 Lot movement request rates 39
4.2 Simulation trial runs 48
4.3 Dataset time steps 51
4.4 Generated training data part 1 54
4.5 Generated training data part 2 55
4.6 Generated training data part 3 56
4.7 Generated training data part 4 57

5.1 Dataset shapes 63
5.2 Model training times 68
5.3 Model performance 69
5.4 Model performance at different prediction horizons 70

Chapter 1

Introduction

Modern semiconductor manufacturing is carried out by a complex automated system in a semiconductor fabrication plant (Fab). To meet the rising demands of the semiconductor market, semiconductor manufacturers have adopted a unified Fab layout approach [33]. In a unified Fab layout, many independent semiconductor processing tools are all interconnected by an overhead hoist transport (OHT) system. An OHT system consists of vehicles that travel on guided rails hung from the ceiling, as pictured in Figure 1-1. A unified Fab layout allows for programmable wafer production, so that the same facility is capable of producing many semiconductor wafer designs and different sequences of processing steps without additional tools.

However, a unified Fab layout means that all the vehicles in the Fab must share the same OHT system. Under high production conditions, there is an increased risk of traffic in the OHT system. Traffic increases production latency and reduces throughput, which is undesirable to semiconductor manufacturers. In the worst case, high production latency may ruin semiconductor wafer lots because certain processing steps are time-sensitive. To further complicate the problem, hundreds of processing steps may be required to produce a single semiconductor wafer lot. Traffic routing in a Fab is controlled by its automated material handling system (AMHS).

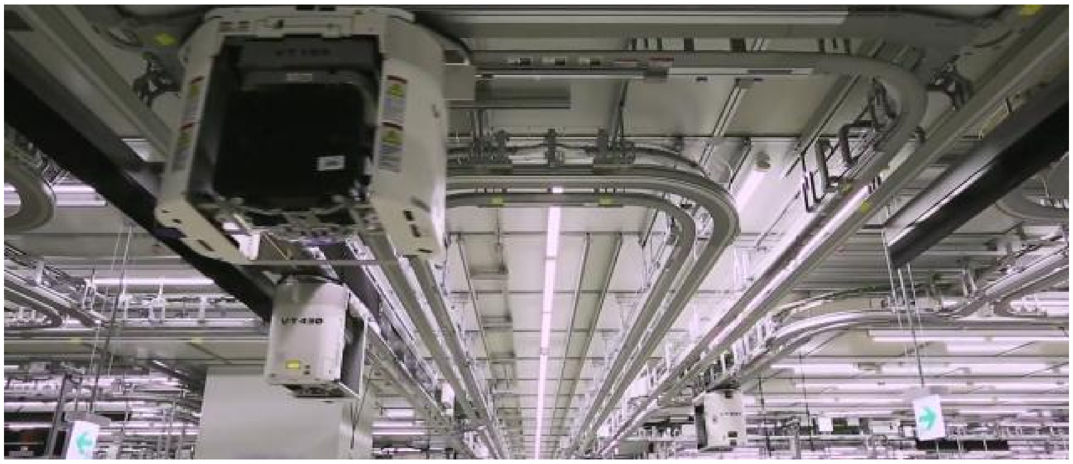

Figure 1-1: A vehicle traveling along the OHT system of a Fab. Image courtesy of Samsung.

1.1 The Task of an AMHS

To make the problem of material handling more approachable, we may divide the task of an AMHS into several stages:

1. The selection of a destination tool from a set of candidate tools, any one of which would be capable of completing the next processing step for the wafer lot.

2. The low-level vehicle programming which controls how the vehicle acquires and deposits the wafer lot and interacts with the tools.

3. The vehicle selection and routing to transport the wafer lot to its destination.

In this thesis, we focus on the traffic routing problem; for recent advances in using machine learning for destination tool selection, we refer the reader to [24].

Improving the traffic routing efficiency of an AMHS increases the utilization of existing hardware and decreases production latency, which is desirable to semiconductor manufacturers. Heuristic-based AMHS traffic routing algorithms are often used in practice [3, 7, 38], but these leave significant opportunities for improvement. Therefore, optimizing AMHS traffic routing has received increasing attention with the rise of deep learning methods. Reinforcement learning methods can achieve strong traffic control performance [23, 19, 21], but they struggle with impractical compute requirements and the narrow scope of the resulting model.

Similarly, heuristic-based or narrow-scope methods may be applied to anomaly detection in AMHS [41]. However, in this project, our goal is a unified framework for spatiotemporal modeling on large graphs. Spatiotemporal modeling has applications in environmental science, social media, traffic forecasting, crime, health care, and other complex system problems [2]. Spatiotemporal modeling also provides general methods for network-constrained trajectory clustering which may be used for anomaly detection [22, 14, 10, 37].

Deep learning methods have been used for spatiotemporal modeling [43], but most models are computationally infeasible on large graphs [28, 45]. METR-LA and PEMS-BAY, two benchmark datasets for spatiotemporal modeling, have 207 and 325 nodes respectively [28]. In contrast, transport graphs with $\sim$10000 nodes render the requisite $\mathcal{O}\left(N^2\right)$ computations infeasible.

1.2 Related Works

Our discussion of related works is split into two parts. First, we discuss spatiotemporal graph neural networks in Section 1.2.1. The models referenced in Section 1.2.1 are nearest neighbors to the Partition WaveNet model that we describe in this thesis. The related works in Section 1.2.1 are specific to the task of spatiotemporal modeling. Second, we discuss dimensionality reduction techniques in Section 1.2.2. Section 1.2.2 contains a more broad overview of the approaches available to reduce the size of graphs. Some of these techniques are specific to neural networks, while others are more general.

1.2.1 Spatiotemporal Graph Neural Networks

Most spatiotemporal graph neural networks use graph convolutions to capture spatial information, and RNNs or CNNs to capture temporal information [44]. Graph convolutions are a method for aggregating node feature information using the edges

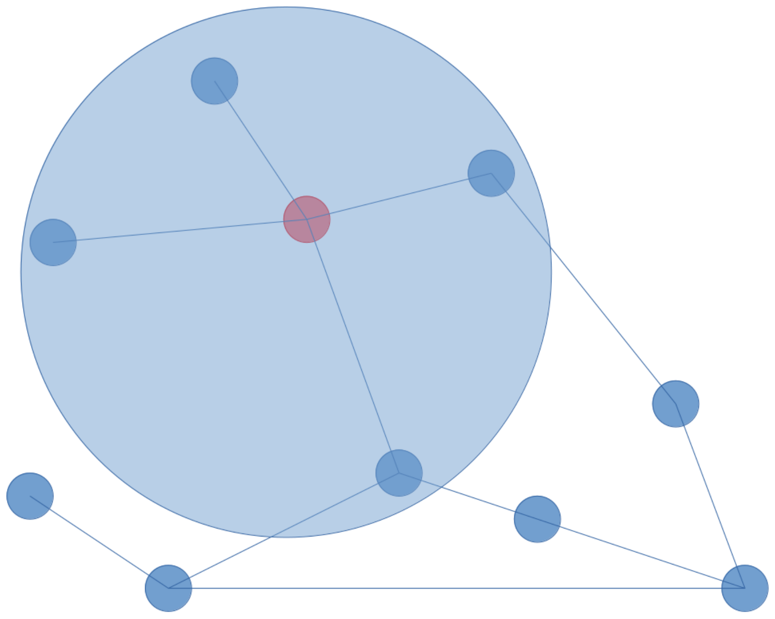

Figure 1-2: A schematic of a graph convolution. For each node in the graph, feature information from its neighbors is aggregated to derive the output.

incident to a node. The nodes involved in an example graph convolution are circled in Figure 1-2. One early RNN-based approach used a mixture of RNNs structured as a graph [20]. Other RNN-based approaches have used diffusion-convolution [28] or gated attention mechanisms [49] to combine the spatial and temporal components. RNN-based approaches tend to suffer from exploding gradients and are slow for long input sequences. CNN-based approaches have combined graph convolutions with 1D temporal convolutions, either using standard [46] or dilated [45] causal convolutions. These approaches are fast and enjoy stable gradients, but they assume it is feasible to perform computations on the $N \times N$ adjacency matrix of the input graph.

1.2.2 Dimensionality Reduction

A graph partition is a well-established method for reducing the size of a graph [4]. In a graph partition, nodes are grouped together into supernodes. Figure 1-3 depicts a graph partition with two supernodes, colored red and blue. The goal of a graph partition is to have few edges between nodes that belong to different supernodes.

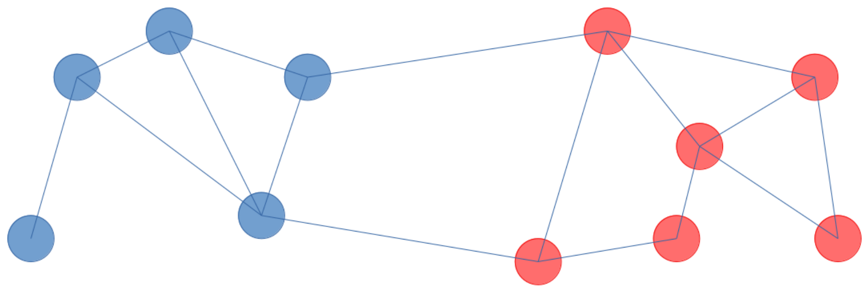

Figure 1-3: An example graph partition that intuitively corresponds to a well-chosen graph reduction.

Then, the supernodes approximate a "zoomed out" view of the adjacency structure present in the original graph.

Graph partitions have been applied successfully to image and video segmentation tasks in machine vision [11]. The graph partition problem is NP-complete, so heuristic-based and deep learning methods are used in practice to find an approximate solution [32]. Graph neural networks have used heuristic-based graph partitions for dimensionality reduction [29], but not in an end-to-end fashion. We seek an end-to-end approach that simultaneously learns an appropriate graph partition and carries out spatiotemporal modeling on the induced representation.

A network embedding may also be used to reduce the size of the graph [8]. In a network embedding, such as the network embedding depicted in Figure 1-4, nodes are mapped to points in $\mathbb{R}$-space. The goal is for close nodes under the adjacency (spatial) structure of the original graph to correspond to close points in $\mathbb{R}$-space.

This approach has been used for anomaly detection in computer networks [17]. While network embeddings offer an end-to-end solution, the adjacency structure from the original graph is lost, and thus deep temporal models may be used with a standard network embedding instead of the desired spatiotemporal models.

For learning graph embeddings, deep neural networks have used random walks [5] and variational methods [26]. Later improvements to the variational methods incorporated adversarial regularization [35, 48]. However, graph embedding approaches lose the original graph adjacency structure. Therefore, they cannot be used as a pre-

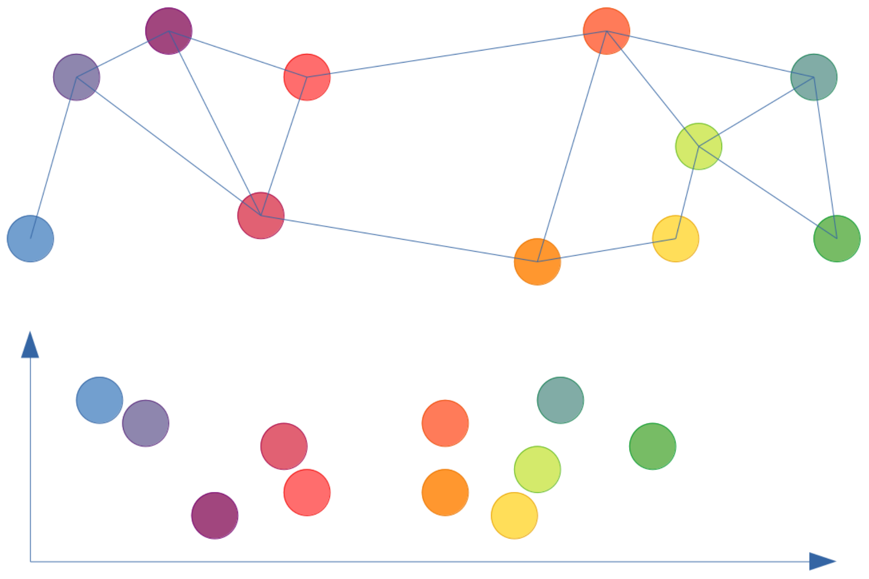

Figure 1-4: A graph and an example network embedding in $\mathbb{R}^2$.

cursor to state-of-the-art spatiotemporal graph networks, which rely on the inputted graph adjacency structure to compute graph convolutions. We desire a method that fully leverages the known spatial information captured in the adjacency matrix of the OHT system graph.

Thus, we seek a graph reduction that can be incorporated into existing neural network architectures in an end-to-end manner, and retains important information about the original graph adjacency structure. We propose Partition WaveNet, a modification of Graph WaveNet [45] for large graphs. Partition WaveNet learns a normalized node embedding, which may be thought of as a “fuzzy” partition, to reduce the graph and render spatial computations feasible.

1.3 Thesis Organization

This thesis is structured as follows. In Chapter 2, we discuss the motivation for our choice to frame the AMHS traffic problem as a supervised learning task. The intuition behind Partition WaveNet and its benefits over alternative neural network architectures are summarized. Next, we explain the mathematical formulation of the problem and our proposed solution in Chapter 3. Then, we explore the implementation details of our simulated Fab and data generation process in Chapter 4. Last, we discuss our experimental results and potential follow-up research in Chapter 5.

Chapter 2

Problem Framing

Broadly, the scope of our problem is to optimize traffic routing in an AMHS. In the context of machine learning, there are two natural paradigms which may be used to frame our problem: reinforcement learning and supervised learning.

2.1 Reinforcement Learning

In reinforcement learning, an agent takes actions to move between states, with the goal of maximizing reward. The agent has multiple attempts to learn an effective policy, i.e., strategy. To specialize reinforcement learning to our problem, we could adopt the following assignments:

- The agent would be an AMHS.
- The actions would be the traffic routing decisions, including which vehicle to assign and which path to take.
- The state would be the positions of the vehicles and semiconductor wafer lots in the OHT system.
- The reward would be a metric for how effectively the AMHS maximizes throughput and minimizes latency.

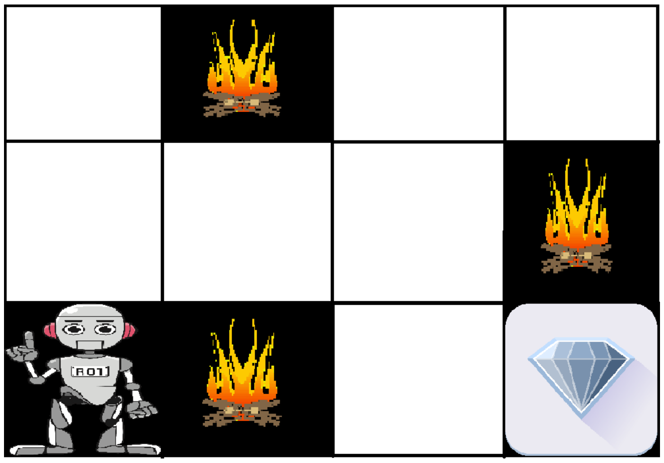

Figure 2-1: A schematic of a toy reinforcement learning problem. The robot agent must take actions to traverse the graph. The diamond represents positive reward, and the fires represent negative rewards.

Alternative assignments are possible. In [19], nodes in the OHT system graph are modeled as agents, and the actions correspond to routing decisions for the vehicles passing through a node. Our arguments against reinforcement learning are not based on these assignments, but rather on general issues that arise in reinforcement learning. Figure 2-1 sets up a toy reinforcement learning problem that we will repeatedly reference to explain these general issues with reinforcement learning.

To make the problem tractable, we would restrict our action space to include only a small subset of possible routing decisions. We would select the actions with the greatest impact on the resulting traffic routing in the OHT system. For example, we might let actions control the edge weights for a shortest path search in the OHT system graph. This shortest path search would determine our vehicle routing policy.

Reinforcement learning is a natural paradigm to consider for the problem of AMHS traffic control, and it has several attractive benefits. In reinforcement learning, we can optimize directly for a reward function that we decide is important. Reinforcement learning is more likely to succeed in settings where data generation is inexpensive, which is the case for AMHS traffic control. Through deep reinforcement learning, we can leverage the power of deep neural networks.

However, we argue that reinforcement learning is less suitable for traffic control in an AMHS than approaches derived from supervised learning. First, reinforcement learning tends to suffer from sample inefficiency [16]. State-of-the-art reinforcement learning agents can require thousands of CPU hours of training to match the results of human agents. In contrast, ground truth- or simulation planning-based models can achieve comparable performance with zero training [42].

Second, reinforcement learning can adopt undesirable policies if the reward function is misspecified. Called "reward hacking," this problem refers to reinforcement learning agents that learn to exploit the reward function rather than learning a desirable policy [1]. In Figure 2-1, reward hacking could correspond to the robot agent never moving in order to collect a passive reward for staying alive.

Reward hacking is particularly dangerous in our problem because our selected reward function must balance the tradeoff between maximizing throughput and minimizing latency. If our reward function is misspecified, then reward hacking may cause our agent to ignore particular lots in pursuit of maximizing throughput, or to over-prioritize particular lots at the expense of overall throughput. In the worst case, reward hacking could cause our agent to never finish processing a lot which is taking too long in order to avoid a negative reward associated with completing the lot. In this way, a well-intentioned reward function has the potential to backfire in edge cases.

Another aspect of the reward function that has potential for misspecification is the simulated environment. The sample inefficiency of reinforcement learning prevents us from training an agent on the production environment, so we would need to train our policy on a simulated Fab. Our simulation is therefore part of our reward function, and any inaccuracies, approximations, or bugs in the simulation have the potential to be exploited via reward hacking. Due to the vast complexity of a modern Fab, this is a significant concern for AMHS traffic control. In Figure 2-1, this type of reward hacking could correspond to the robot agent traveling out of bounds to navigate the maze and collect the diamond, if this behavior were mistakenly allowed by the simulation environment.

Reward hacking aside, reinforcement learning has the added downside of requiring the reward function to be specified before training the model. This is undesirable

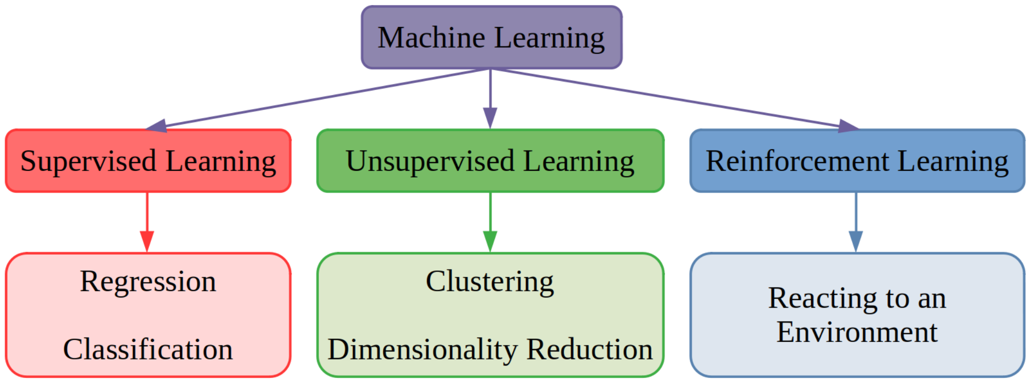

Figure 2-2: A schematic relating different types of machine learning. The bottom row lists problems to which each type of machine learning would be applied.

in the setting of AMHS traffic control because as previously mentioned, we have two, possibly competing goals: maximizing throughput and minimizing latency. In a modern Fab, which fulfills many lot requests simultaneously, the desired balance between these two rewards may change over time. In an extreme case, semiconductor manufacturers may wish to complete a high-priority lot as quickly as possible and maximize throughput on the other lots. We would prefer the flexibility of changing our reward function in a production model and having ready access to a corresponding policy that maximizes the new reward function. To achieve this flexibility, we turn to supervised learning approaches.

2.2 Supervised Learning

In supervised learning, we seek to approximate a function based on a limited number of input-output pairs, which we call our training data. This paradigm has the potential drawback of requiring high-quality training data for the function we would like to approximate. Nonetheless, we find that supervised learning is well-suited for the AMHS traffic control problem. An overview of the high-level machine learning fields, including both reinforcement learning and supervised learning, is shown is Figure 2-2.

To specialize supervised learning to our problem, we would need to select a function to approximate. In AMHS traffic control, there are several functions we may consider approximating:

- f_1 : system state $\mapsto$ desired vehicle routing policy
- f_2 : (system state, vehicle routing policy) $\mapsto$ improved vehicle routing policy
- f_3 : system state $\mapsto$ future system state
- f_4 : (system state, vehicle routing policy) $\mapsto$ future system state

While f_1 or f_2 would be ideal, learning a function that outputs a desired vehicle routing policy would be infeasible without high-quality training data. Accordingly, this would require access to a source of effective vehicle routing policies, which would defeat the purpose of our model. Therefore, we focus our attention on the functions that output the future system state.

Because f_3 implicitly assumes an underlying vehicle routing policy, we seek to approximate a fourth function instead, f_4: (system state, vehicle routing policy) $\mapsto$ future system state. This is a generalization of f_3. Note that we are using our definition of system state from Section 2.1: the positions of the vehicles and semiconductor wafer lots in the OHT system. Depending on our access to the internal states of the tools in the Fab, we may augment our system state with this data accordingly. As in Section 2.1, we consider a restricted, parameterized action space: the vehicle routing policies corresponding to a shortest path search using selected edge weights.

We now contrast this approach with a reinforcement learning-based approach. Instead of training an agent to make traffic routing decisions for the AMHS as we would do in reinforcement learning, we are training a simulation model that predicts future traffic given a vehicle routing policy and the current vehicle and semiconductor wafer lot positions. Unlike reinforcement learning, supervised learning can be quite sample efficient. Supervised learning can approximate the future system state well with limited training data, which is important in our problem because the OHT system graph is large. So we expect supervised learning to be tractable when reinforcement learning would not be tractable.

Supervised learning allows for a wider variety of applications than would reinforcement learning. Whereas reinforcement learning would only optimize one prespecified reward function, supervised learning allows for a dynamic reward function that can be changed after training. For example, given a reward function on system states, we could use Monte Carlo Tree Search to optimize the reward function evaluated at the end of some desired time horizon. Alternatively, we could detect Fab-wide anomalies by measuring the difference between the predicted system state and reality. If many vehicles are out of place according to our model prediction, then we may suspect a disruption to normal Fab operation.

In order to generate training data for our supervised learning model, we must specify our vehicle routing policy in advance. We would not like to restrict ourselves to just one vehicle routing policy, so to circumvent this limitation, we generate training data that is a mix of many simulated runs all using different vehicle routing policies. We discuss our training data generation process further in Chapter 4.

Now, we may ask the question "if our model is just an approximation of our simulation environment, then what value does it add?" We argue that in fact, we should hope to recover our simulation environment in the limit of infinite training data. If we perfectly learned the dynamics of our simulation environment, then the problem of finding an effective vehicle routing policy would be reduced to a tree search problem, which could then be optimized by any number of methods [15].

Moreover, approximating our simulation environment has desirable denoising effects in the finite-sample case. Based on limited training data, we do not learn the extreme edge cases of our simulation environment, but rather the general trends and behaviors according to a compressed latent space for our simulation environment. Consequently, we are less likely to fall prey to reward hacking or the exploitation of loopholes in our simulation environment. There is also the benefit of computation time. We expect the model learned by supervised learning to run more quickly than the underlying full simulation model. This makes the tree search to find an effective vehicle routing policy more efficient.

2.3 WaveNet in Practice

WaveNet is a CNN-based architecture designed for supervised learning tasks involving time-series data [34]. WaveNet has many benefits over RNN-based predecessors for time-series analysis: faster training, stable gradients, and a larger receptive field size. The faster training of CNN-based approaches is a necessity for modeling complex systems as in our problem, and the large receptive field size means that WaveNet is better able to model OHT system dynamics over longer time scales. This is important because a vehicle traveling from one side of the OHT system to another takes many units of time to complete its journey. So only a neural network architecture with a large receptive field size, such as WaveNet, would be able to effectively model traffic over longer time scales.

Given a high-fidelity traffic model approximation as would be provided by a trained WaveNet model, the task of traffic route optimization in an AMHS is greatly simplified. We must only apply penalties to edges in accordance with the level of predicted traffic at each edge. If we produced traffic predictions at multiple time scales, then we could apply diminishing penalties to traffic predictions farther in the future. In this way, vehicles could be rerouted to minimize future traffic. This summarizes one way that an effective and flexible vehicle routing policy could be derived from a trained traffic model.

Chapter 3

Methods

In this chapter, we begin by outlining our problem formulation. Then, we describe the building blocks of our proposed network architecture, focusing on our novel partition-embedded graph convolution layer. Last, we present our proposed architecture for deep spatiotemporal modeling on large graphs.

3.1 Problem Formulation

We follow a similar problem formulation to that of Graph WaveNet [45]. We are given a graph $G = (V, E)$ with nodes V and directed edges E, and we suppose that $|V| = N$ is large. The graph adjacency matrix is denoted by $\mathbf{A} \in \mathbb{R}^{N \times N}$, where $\mathbf{A}_{i,j} = \mathbf{1}\{(v_i, v_j) \in E\}$. We operate in the setting where G is a sparse graph with a convenient sparse representation for $\mathbf{A}$. At time step t, we observe graph signals $\mathbf{X}^{(t)} \in \mathbb{R}^{N \times D}$. Given a graph and its S historical graph signals, our goal is to forecast its next T graph signals:

$$\left[\mathbf{X}^{(t-S):t}, G\right] \xrightarrow{f} \mathbf{X}^{(t+1):(t+T)} \;, \tag{3.1}$$

where $\mathbf{X}^{(t-S):t} \in \mathbb{R}^{N \times D \times S}$ and $\mathbf{X}^{(t+1):(t+T)} \in \mathbb{R}^{N \times D \times T}$. We suppose that $D, T \ll N$ so that computations involving the graph signals are feasible, but computations involving the adjacency matrix are not.

For large N and potentially large S, we must take care to note the format of the data that we would like to model. We might expect the adjacency matrix $\mathbf{A}$ and the graph signals $\mathbf{X}^{(t)}$ to both exist in compressed representations. For the graph signals, this could be lists of lot and vehicle trajectories with corresponding timestamps. If this is the case, expanding the dataset to $\mathcal{X} \in \mathbb{R}^{N \times D \times S}$ may also be infeasible. For our problem formulation to apply in this setting, we would need to coarsen the time dimension of the input data by grouping together consecutive timestamps and aggregating the corresponding graph signals. For intuitive physical signals like lot count, vehicle count, traffic congestion, etc., this aggregation is straightforward, and dimensionality reduction via the time dimension is simpler than via the spatial dimension.

3.2 Partition-Embedded Graph Convolution Layer

The graph convolution layer is defined as

$$\mathbf{Z} = \tilde{\mathbf{A}}\mathbf{X}\mathbf{W} \ , \tag{3.2}$$

where $\mathbf{Z} \in \mathbb{R}^{N \times M}$ is the output, $\tilde{\mathbf{A}} \in \mathbb{R}^{N \times N}$ is the normalized adjacency matrix with self-loops, and $\mathbf{W} \in \mathbb{R}^{D \times M}$ is the model parameter matrix [25]. We use a node embedding dictionary with learnable parameters $\mathbf{E} \in \mathbb{R}^{N \times b}$ to reduce the N dependence of our computation. For our node embedding dictionary, we propose

$$\mathbf{B}_{emb} = \mathrm{SoftMax}(\mathbf{E}) \ . \tag{3.3}$$

The SoftMax function normalizes the embedding, which ensures that a unit graph signal in G maps to a unit graph signal under the embedding. This is important in AMHS, where a graph signal may correspond to lot or a vehicle spatiotemporal observation. We would like to enforce the conservation of these graph signals. Note that $\mathbf{B}_{emb}$ defines a continuous partition of G in which nodes are permitted to have partial group assignments. Using $\mathbf{B}_{emb}$, we may define the embedded graph signal as

$$\mathbf{X}_{emb} = \mathbf{B}_{emb}^{T}\mathbf{X} \ . \tag{3.4}$$

We proceed by using $\mathbf{X}_{emb} \in \mathbb{R}^{(b \times D)}$ in place of $\mathbf{X} \in \mathbb{R}^{(N \times D)}$ in order to lessen the computational burden of large N. The adjacency matrix $\mathbf{A}$ is replaced by $\mathbf{A}_{emb} = \mathbf{B}_{emb}^{T}\mathbf{A}\mathbf{B}_{emb}$. The resulting layer, which we call a partition-embedded graph convolution layer, is given in Equation 3.5:

$$\mathbf{Z} = \tilde{\mathbf{A}}_{emb}\mathbf{X}_{emb}\mathbf{W} \ , \tag{3.5}$$

where $\tilde{\mathbf{A}}_{emb} = \mathbf{A}_{emb}$ with normalization and self-loops added. Equations 3.2 and 3.5 are equally feasible using a sparse representation for $\mathbf{A}$. However, we would like to incorporate diffusion-convolution [28] and a self-adaptive adjacency matrix [45]. Diffusion-convolution takes the power series of the transition matrix $\mathbf{A}/\text{rowsum}\left(\mathbf{A}\right)$ to aggregate information from successively larger orders of neighborhoods in the graph. This is not feasible for large N and sparsely-represented $\mathbf{A}$, but it is feasible for our embedded adjacency matrix $\mathbf{A}_{emb}$.

While self-adaptation of our full adjacency matrix is not essential for a known and fixed OHT system graph, we desire our approach to adapt $\mathbf{A}_{emb}$ for applicability to wider scenarios, and to adapt in response to approximations resulting from the partition embedding. Even for a known and fixed OHT system graph, self-adaptation gives our model the flexibility to capture coincidental long-distance spatial dependencies that are stronger than our adjacency matrix would otherwise predict.

Therefore, we modify our partition-embedded graph convolution layer to incorporate diffusion-convolution and a self-adaptive adjacency matrix. Overall, we have

$$\mathbf{Z} = \sum_{k=0}^{K} \mathbf{P}_{emb}^{k}\mathbf{X}_{emb}\mathbf{W}_{k} \ , \tag{3.6}$$

where $\mathbf{P}_{emb}^{k}$ represents the power series of the embedded transition matrix $\mathbf{A}_{emb}/$

rowsum $(\mathbf{A}_{emb})$. This may be adapted to directed graphs by replacing $\mathbf{P}_{emb}$ with a forward and a backward transition matrix. We perform this replacement for all our models in Chapter 5 because the OHT system graph is directed.

3.3 Temporal Convolution Layer

For modeling node temporal trends, we use a stacked dilated 1D causal convolution [47] based on the WaveNet architecture [34]. A dilated causal convolution with zero-padded inputs computes a standard 1D convolution, except that it samples inputs with period given by the dilation factor and skips the rest. The dilated causal convolution operator is described by

$$(\mathbf{x} \star \mathbf{f})(t) = \sum_{s=0}^{K-1} \mathbf{f}(s)\mathbf{x}(t - d \cdot s) \ , \tag{3.7}$$

where d is the dilation factor, $\mathbf{x} \in \mathbb{R}^T$ is the 1D input sequence, and $\mathbf{f} \in \mathbb{R}^K$ is the convolution filter. The relationship between the receptive field size and the number of hidden layers is illustrated in Figure 3-1.

Gating mechanisms [9] are a simple but powerful approach for controlling information flow in RNNs and in temporal CNNs. The gating mechanism of a temporal convolution layer is described by

$$\mathbf{h} = g(\mathbf{\Theta_1} \star \mathcal{X} + \mathbf{b}) \odot \sigma(\mathbf{\Theta_2} \star \mathcal{X} + \mathbf{c}) \ , \tag{3.8}$$

where $\mathcal{X} \in \mathbb{R}^{N \times D \times S}$ is the input and $\mathbf{\Theta_1}$, $\mathbf{\Theta_2}$, $\mathbf{b}$, and $\mathbf{c}$ are model parameters. Here $\odot$ is the entrywise product, $g(\cdot)$ is the activation function applied entrywise, and $\sigma(\cdot)$ is the sigmoid function applied entrywise. The sigmoid function serves as the gate, and it controls the ratio of information passed to the next layer.

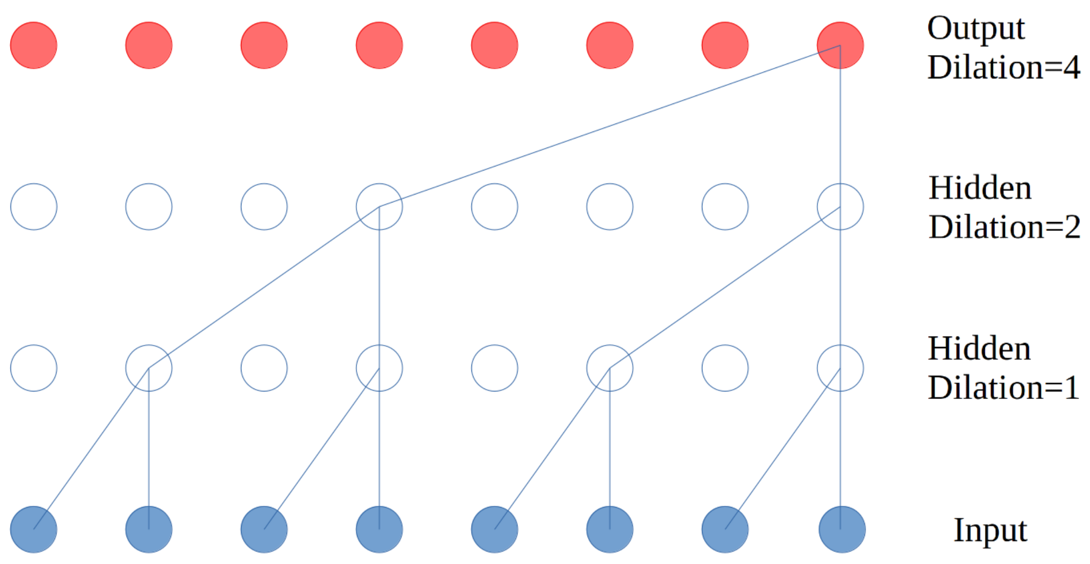

Figure 3-1: A schematic of a stacked dilated 1D causal convolution. The receptive field size grows exponentially with the number of hidden layers, enabling the architecture to efficiently capture long time scale temporal dependencies in the data [47, 34].

3.4 Architecture

For our neural network architecture, we adopt the Graph WaveNet architecture [45] with our proposed Partition-Embedded GCN layers in place of the Graph WaveNet GCN layers. The Graph WaveNet architecture is depicted in Figure 3-2.

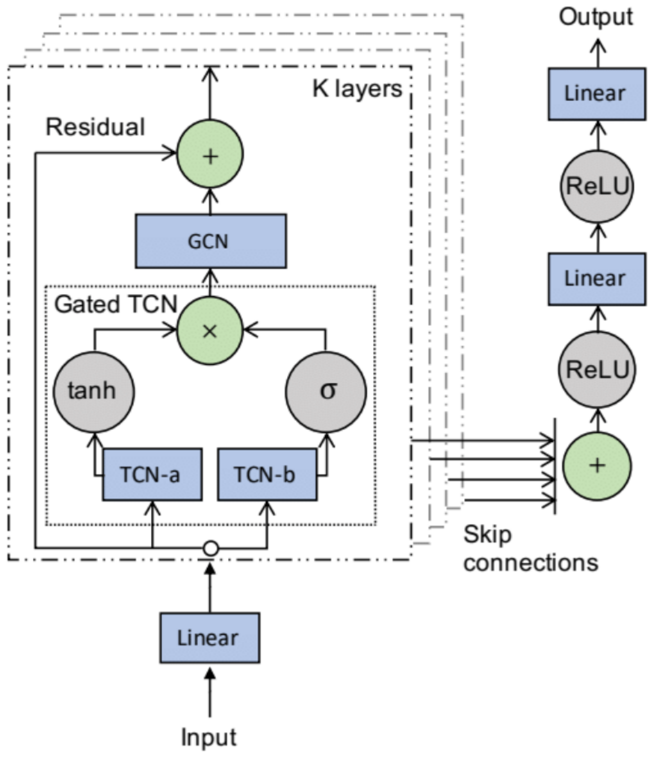

Figure 3-2: A schematic of the Graph WaveNet architecture as featured in the original Graph WaveNet paper [45]. The K spatiotemporal layers are stacked using increasing dilation factors as illustrated in Figure 3-1.

Chapter 4

Simulation

This chapter is dedicated to our implementation of a simulated Fab. First, we discuss what we desire from our simulation, as these goals affect our design choices. Next, we go into the specifics of our implementation, difficulties with deadlock, and design abstractions. Then, we present statistics about our Fab simulation. Last, we describe how we derive training data from our simulation. Our codebase is available at `https://github.com/david-amirault/amhs`.

4.1 Goals

The fundamental purpose of our Fab simulation is to define an environment for evaluating AMHS vehicle routing policies. While this remains the primary goal of our simulation, we identify a number of subsidiary and additional goals in the following list of desiderata:

D.1 Realistic: the simulation environment should accurately reflect the dynamics of the interactions between vehicles, lots, and tools in a real-world Fab.

D.2 General: the simulation environment should support arbitrary OHT system graphs, vehicle travel times, lot acquire and deposit times, and tool processing times.

D.3 Expressive: the simulation environment should enable the user to easily create different vehicle routing policies.

D.4 Visual: we should be able to visualize vehicle movement under different routing policies using our simulation environment.

D.5 Random: the simulation environment should use a statistical generative model for lot requests.

D.6 Repeatable: we should be able to duplicate sequences of lot requests in order to compare different vehicle routing policies using exactly the same lot requests.

D.7 Nontrivial: we should be able to create difficult to solve AMHS traffic control problems using our simulation environment.

D.8 Fail-safe: it should be impossible for a vehicle routing policy to break the simulation environment.

D.9 Standard: when possible, our simulation environment should adhere to existing benchmarks and conventions in the field.

D.10 Free and open source: we would like to publish our simulation environment for others in the community to freely use.

We reference these desiderata when we discuss our corresponding design choices in Section 4.2.

4.2 Implementation

The most prevalent Fab simulation environment in the semiconductor manufacturing industry is Applied Material's AutoMod$^{\text{TM}}$. However, in pursuit of D.10, we would like to avoid using proprietary software as part of our simulation environment. Therefore, we create our simulation environment using SimPy, a process-based discrete-event simulation framework programmed in Python [6]. Other frameworks

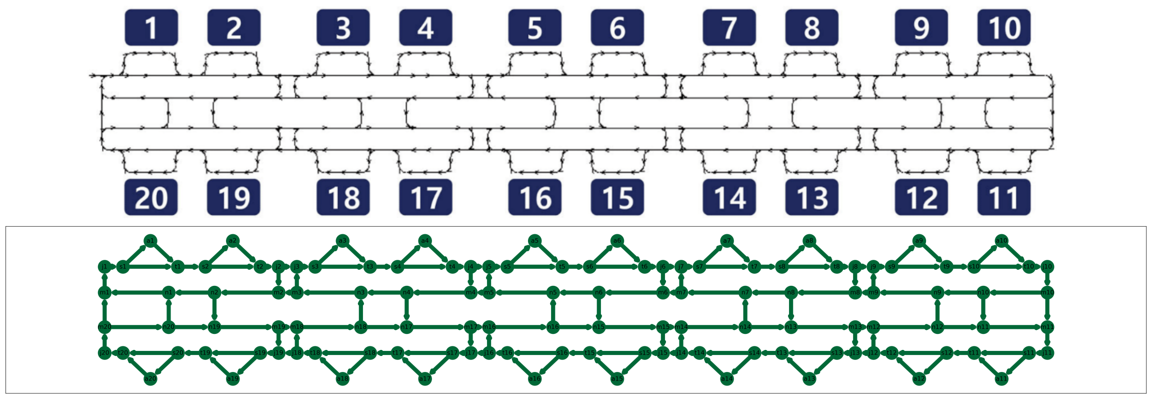

Figure 4-1: The default OHT system graph layout. The top image is the original layout created in [19] using Applied Material's AutoMod^{TM}, and the bottom image is our recreated version in SimPy. The labels 1–20 represent tools in the Fab.

for Fab simulation are possible [30, 27], but SimPy has the advantage of being free and open-source.

Through advanced usage of Python generators, SimPy is able to simulate concurrent process interactions despite the Python global interpreter lock, which enforces single-threaded code execution. SimPy offers low-level shared resource primitives for synchronizing the simulated concurrent processes. These shared resource primitives are general, so we implement the AMHS-specific structures from the ground up: vehicles, lots, tools, and the OHT system. We examine our design abstractions for these AMHS-specific structures in Section 4.4.

To maintain standardization when possible per D.9, we select defaults for our environment drawn from existing literature. For D.2, our default OHT system graph has the same layout as used by [19]. A schematic of the layout is provided in Figure 4-1. We infer the vehicle travel times and lot acquire and deposit times from a video of the simulation in action. Our simulated Fab has 20 tools, which we name a1–a20.

For D.5, our statistical generative model for lot requests uses the same generative model defined by [19]. This generative model is based on a Poisson process, as is standard in manufacturing. Lot requests are generated independently for each pair of tools. Therefore, we choose not to model the hundreds of potential processing steps which may be required to complete the processing for one lot. Instead, we model each pairwise movement request as a separate lot. For the purpose of AMHS vehicle control, the two are equivalent, so this is a convenient simplifying assumption to make. The expected number of requests generated for each pair of tools over one day is presented in Table 4.1.

We use discrete-time simulation instead of continuous-time simulation. Our rationale is to promote ease of visualization per D.4, and to increase the complexity of the AMHS traffic control problem per D.7. In a discrete-time simulation, we frequently find that vehicles, lots, and tools become available simultaneously, which facilitates the creation of complex, multifaceted decisions. It is easier to create Fab visualizations frame by frame in a discrete-time simulation. For our simulation, one unit of time represents one second. To implement a Poisson process in our discrete-time simulation, we may sample independent Bernoulli random variables at each time step to decide whether or not to create a lot request. The success probabilities of the Bernoulli random variables are given by the lot request rates in Table 4.1 normalized by the number of seconds per day.

For efficient graph algorithm implementations and graph visualization tools, we employ the Python NetworkX package [13]. Using NetworkX, we can initialize a weighted, directed graph from a list of edges and efficiently carry out an all pairs shortest path computation. We use NetworkX graph visualization utilities to update node and edge colors based on current traffic levels for our D.4 vehicle movement visualization.

4.3 Preventing Deadlock

Our desiderata from Section 4.1 place a complex set of constraints on what we would consider to be a suitable simulation environment. One notable issue that our simulation must overcome is deadlock. In this section, we explain why deadlock is a recurring issue for our simulation environment and how we can prevent it.

By D.7, our simulation environment must allow traffic to jam. Without traffic jams, our simulation environment would be incapable of producing useful training data because the AMHS traffic control problem would be trivial. Moreover, by D.1,

Table 4.1: Lot movement request rates for our statistical generative model for lot requests. Movement rates are in lots/day. For this data, we credit [19].

	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
a1	0.0	0.0	0.0	116.1	161.6	0.0	0.0	85.4	37.4	2.7
a2	0.0	0.0	158.2	32.1	0.0	87.4	85.4	0.0	0.0	40.0
a3	0.0	89.8	0.0	0.0	0.0	158.2	77.1	0.0	0.0	55.3
a4	84.2	0.0	0.0	0.0	116.1	32.1	0.0	77.1	81.8	34.5
a5	311.8	18.2	0.0	158.5	0.0	0.0	9.3	298.9	278.2	7.7
a6	0.0	288.0	220.1	47.9	48.3	0.0	378.1	88.5	44.4	319.2
a7	2.3	70.6	206.2	0.0	14.7	568.5	0.0	0.0	109.4	0.0
a8	72.9	0.0	0.0	206.2	471.8	111.5	0.0	0.0	109.4	0.0
a9	37.0	0.2	0.0	97.0	318.4	53.5	0.0	35.5	0.0	0.0
a10	0.0	41.4	111.7	1.3	0.0	394.9	256.2	220.7	0.0	0.0
a11	0.0	72.9	113.8	92.4	0.0	583.3	0.0	0.0	0.0	109.4
a12	72.9	0.0	0.0	7.9	670.2	111.5	0.0	0.0	15.2	94.2
a13	46.3	5.1	0.0	0.0	46.3	5.1	19.1	62.5	435.2	26.4
a14	0.0	41.1	0.0	0.0	0.0	41.1	43.4	0.0	0.0	378.7
a15	0.0	0.0	2.0	0.0	0.0	5.9	257.5	0.0	0.0	0.0
a16	0.0	0.0	0.0	1.1	6.8	0.0	0.0	30.5	271.6	0.0
a17	0.0	0.0	0.0	0.0	4.6	0.0	0.0	0.0	0.0	0.0
a18	0.0	0.0	0.3	0.9	0.1	3.5	363.3	0.0	0.0	0.0
a19	0.0	0.0	2.0	0.0	0.0	5.9	0.0	0.0	0.0	310.4
a20	0.0	0.0	0.0	2.0	5.9	0.0	0.0	590.3	0.0	0.0

	a11	a12	a13	a14	a15	a16	a17	a18	a19	a20
a1	0.0	85.4	138.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
a2	85.4	0.0	15.4	123.4	0.0	0.0	0.0	0.0	0.0	0.0
a3	77.1	0.0	0.0	284.3	33.7	0.0	0.0	20.7	18.3	0.0
a4	0.0	2.9	266.5	0.0	0.0	33.7	19.0	0.0	15.4	0.0
a5	0.0	373.1	272.9	19.7	0.0	31.7	18.7	0.0	0.6	65.4
a6	387.5	88.5	23.4	146.3	31.7	0.0	0.0	19.5	31.1	0.0
a7	0.0	0.0	96.4	219.2	0.0	0.0	0.0	17.2	182.2	2.6
a8	0.0	0.0	315.6	0.0	0.0	192.2	0.0	0.0	0.0	9.9
a9	256.2	256.2	191.7	0.0	0.0	17.8	98.7	0.0	2.4	17.8
a10	0.0	0.0	3.5	203.5	17.8	0.0	0.0	112.0	15.5	0.0
a11	0.0	0.0	0.0	315.6	192.2	0.0	0.0	0.0	9.9	0.0
a12	0.0	0.0	315.6	0.0	0.0	0.0	22.5	0.0	0.0	179.6
a13	0.0	62.5	0.0	0.0	0.0	370.4	213.7	0.0	114.0	370.4
a14	62.5	0.0	0.0	0.0	370.4	0.0	0.0	228.0	256.5	0.0
a15	332.8	0.0	14.6	32.9	0.0	0.0	0.0	0.0	0.0	0.0
a16	0.0	288.3	47.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
a17	8.0	332.5	27.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0
a18	0.0	0.0	0.0	29.3	0.0	0.0	0.0	0.0	0.0	0.0
a19	279.9	0.0	0.0	47.5	0.0	0.0	0.0	0.0	0.0	0.0
a20	0.0	0.0	47.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0

traffic must be permitted to jam badly; this is what happens when real-world road systems are burdened with too many vehicles. D.8 requires that no traffic jam may be inescapable, but D.1 implies that there is no immediate and direct way to enforce this.

By the realistic requirement D.1, on any given path in the OHT system graph, the order of the vehicles is fixed. If a vehicle is blocking our way, then we must wait for the vehicle to clear out in order to continue traversing the OHT system. This also means that only one vehicle may occupy an intersection, i.e., node, at a time. These conditions together imply that deadlock is possible on our OHT system graph, and it occurs precisely when all the nodes and edges in a cycle have reached their respective vehicle capacities. Increasing the vehicle capacities is not an option because this would trivialize the problem, which would violate D.7.

Unfortunately, detecting at-capacity cycles is insufficient for addressing deadlock. Figure 4-2 highlights the difficulty of preventing deadlock. We imagine that the cycle depicted in Figure 4-2 currently has capacity for 4 additional vehicles. Seeing this remaining capacity, our AMHS could decide to direct a vehicle along each edge leading into the nodes $m4$, $m16$, $n5$, and $n17$. If these vehicles enter the cycle simultaneously, then the cycle goes directly from capacity $= 4$ to deadlock.

Preventing deadlock requires a preemptive approach. This is because our OHT system graph has cycles, and if a cycle fills up, then it is already too late to stop a deadlock from occurring. Ideally we would cancel a movement request and reroute one of the stuck vehicles to make space, but this is not permissible under SimPy. SimPy uses blocking requests for space to open up in the destination node or edge, so we are dealing with deadlock which occurs in the software as well. Detecting deadlocks preemptively is difficult because our OHT system graph has many cycles, each with multiple incoming edges for us to consider.

Another deadlock-related concern is the runtime of our attempted solution. We found that standard graph algorithms are too slow to be feasible for preventing deadlock. For example, with a runtime of $\mathcal{O}(|V|+|E|)$, breadth-first search would have to perform at least 624 operations on our selected OHT system graph. For 100 simulated

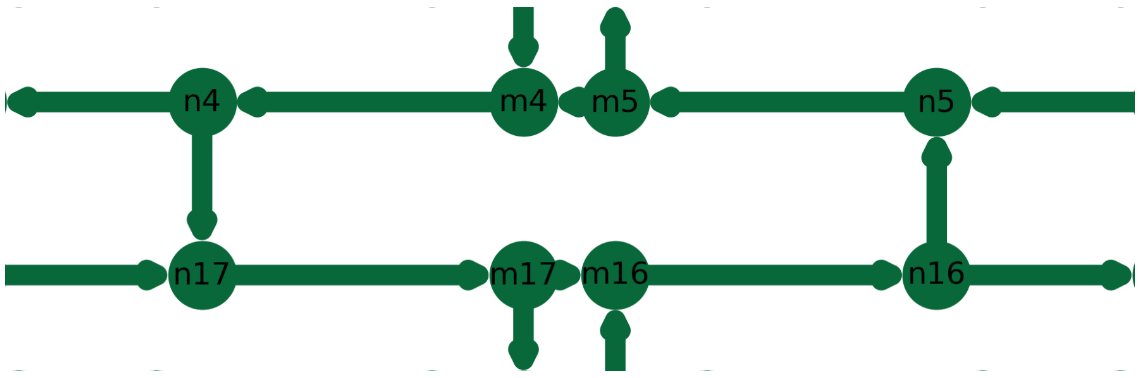

Figure 4-2: A cycle in our OHT system graph that is vulnerable to traffic jams. The four incoming edges to the cycle at m4, m16, n5, and n17 allow a traffic jam to become inescapable before the cycle has reached its vehicle capacity.

runs, each lasting 1 day = 86400 seconds, performing this many operations at each time step would place a huge computational burden on our simulation environment. So we seek algorithms that are sublinear in the size of the OHT system graph.

One approach is to hard code the cycles that are capable of producing deadlock, of which there are four cycles with capacity 20, ten cycles with capacity 25, three cycles with capacity 40, and two cycles with capacity 42. This attempted solution has too many edge cases, so another approach is required.

To fully resolve the deadlock problem, we draw inspiration from real-world traffic systems. In real-world road systems, if there is no space, then additional cars are not allowed to merge onto the road. In this way, completely stopped traffic is prevented. To implement this idea, we create priority requests for the right to travel through a node corresponding to a merge. Vehicles already in the loop take priority over vehicles attempting to merge. This keeps traffic moving around the loops without allowing them to fill up completely. We find that this solution fully resolves the deadlock problem without negatively affecting simulation performance.

By resolving the deadlock problem, we address the primary difficulty of D.8. In the process, we unlock a greater region of the space of vehicle routing policies for our training process to explore. Now that our training process no longer has to worry about deadlock, we can generate training data according to ineffective vehicle routing policies as well. Then, our training data can span a broader region of the space of vehicle routing policies, which has the potential to lead to a more robust supervised learning model.

4.4 Design Abstractions

We implement the AMHS structures using an object-oriented programming approach. This section documents the relevant fields and methods associated with each Python class and how they relate to our desiderata from Section 4.1.

Our AMHS structures use the following SimPy shared resource primitives for synchronization: `resource`, that represents a discrete shared resource supporting prioritized requests, and `store`, that represents a shared resource for storing objects supporting requests for specific objects.

4.4.1 AppSite

The `AppSite` class is a container class that represents a node in the OHT system graph. This class serves the dual-purpose of representing tools in the Fab as well as intersections in the OHT system. An `AppSite` has a `string` field for naming purposes, a SimPy `Resource` field with customizable capacity representing the number of lots the tool can hold, a `PriorityResource` field with capacity 1 for vehicles to request exclusive access to the `AppSite`, and three time fields. The time fields allow the user to customize the time durations required for the tool to process a lot, for a vehicle to acquire or deposit a lot at the `AppSite`, and for a vehicle to travel through the `AppSite`. By default, we set the lot acquisition and deposition times to high values of 15 seconds each to induce traffic jams at the tools in the Fab. This class is the primary means of customization for D.2. This class also has the capability to serve as a side-track buffer, i.e., temporary storage unit for lots in the Fab.

4.4.2 RailPath

The `RailPath` class is a container class that represents an edge in the OHT system graph. A `RailPath` has a `string` field for naming purposes, a SimPy `Resource` field with customizable capacity representing the number of vehicles the edge can hold, and a time field that controls how long it takes for a vehicle to travel through the `RailPath`. This class is the secondary means of customization for D.2. Note that a `PriorityResource` is only needed at the `AppSite`s, where merges occur, in order to prevent deadlock. An ordinary SimPy `Resource` is sufficient here.

4.4.3 FOUP

The `FOUP`, or front-opening unified pod, class represents a semiconductor wafer lot. A `FOUP` has a `uuid` field that stores a unique identifier for the lot, a time field that controls how long to wait at the beginning of the simulation before creating the lot, a location field that stores the name of the node or edge at which the lot is currently located, a status field that describes what the lot is currently doing, and an ordered list of tool demands that gives the remaining processing steps in order to completely process the lot.

The `FOUP` status values that we allow are `WAIT IN`, `TRANSFERRING`, `COMPLETED`, `APPLICATION`, and `WAIT OUT`. The first entry of the demand list gives the starting location of the lot. Because we model each lot movement request as a separate lot, our demand lists all start out with two demands: the starting tool and the ending tool. However, one could use our simulation environment to model lots with hundreds of processing steps if desired. Our simulated runs each have tens of thousands of lot movement requests, so we use the Python uuid package to generate unique identifiers for each corresponding `FOUP`.

The `FOUP` class has a method for completing a processing step if the lot is at the appropriate tool. Per D.8, the `FOUP` class verifies that the lot is at the appropriate tool before it completes the processing step. We use a SimPy `FilterStore` to store all the active lots in our simulation. The AMHS can request a specific lot or any lot from the `FilterStore`. The `FOUP` class logs tool usage times so that the performance of the AMHS vehicle routing policy can be evaluated after the run.

4.4.4 OHT

The `OHT`, or overhead hoist transport, class represents a vehicle. An `OHT` has a `uuid` field that stores a unique identifier for the vehicle, a location field that stores the name of the node or edge at which the vehicle is currently located, a status field that describes what the vehicle is currently doing, and a field that saves a reference to the `FOUP` the vehicle is carrying or `None` if there is no such `FOUP`.

The `OHT` status values that we allow are `ENTERING`, `PARKED`, `ENROUTE`, `ACQUIRING`, `DEPOSITING`, and `REMOVED`. We again use the Python uuid package to generate unique identifiers for each `OHT`. We initialize all our vehicles to enter near tool a1 in the upper-left corner of our OHT system graph, following the precedent set by [19] for D.9. Vehicles are the most significant AMHS structure involved in traversing the OHT system graph, and the `OHT` class has corresponding methods to simulate vehicle activities. The `OHT` class has methods for taking a `RailPath`, acquiring a `FOUP`, and depositing a `FOUP`.

Per D.8, the OHT class guarantees that all attempted edge traversals, lot acquisitions, and lot depositions are valid. In pursuit of D.1, the `OHT` class leverages the SimPy shared resource primitives to guarantee that a vehicle only travels where there is space for it and that the vehicles pass through the intersections in the OHT system graph one at a time. The `OHT` class is responsible for updating lot location and status after the lot is acquired by a vehicle. Like the `FOUP` class, the `OHT` class logs all edge and node traversals for diagnostic and evaluation purposes.

4.4.5 MHS

The `MHS`, or material handling system, class is a container class that represents all of the physical AMHS structures. An `MHS` has fields for the nodes in the graph stored in a hash table that maps node names to `AppSite`s, the edges in the graph stored in a hash table that maps edge names to `RailPath`s, the `FilterStore` of `FOUP`s in the simulated Fab, the `FilterStore` of `OHT`s in the simulated Fab, and the OHT system graph constructed via the Python NetworkX package. The `MHS` class provides a convenient wrapper around the physical AMHS structures.

4.4.6 AMHS

The `AMHS`, or automated material handling system, class is the parent class for all vehicle routing policy implementations. It has a field for the `MHS`, and it may be extended by subclasses to have any number of other fields. The `AMHS` class promotes flexibility in implementing vehicle routing policies, which contributes to D.3. Subclasses of `AMHS` must implement a single method: `run`, which carries out all the vehicle routing policy logic.

4.4.7 Simulation

The `Simulation` class represents one run of our simulated Fab. When initialized, the `Simulation` class creates all lot movement requests for the simulated run. We create the lot movement requests according to the Poisson process generative model described in Section 4.2, in accordance with D.5. The `Simulation` class accepts a random seed argument that allows simulated runs to be repeated per D.6. Alternatively, the `Simulation` class also supports the creation of multiple simulated runs from the same `Simulation` instance for runtime efficiency purposes. We implement an optional burn-in period during which lot request rates are linearly scaled up to their desired levels; this is a convention followed by [19] that we adopt for D.9.

For space efficiency purposes, the `Simulation` class uses a sparse representation for storing the generated lot movement requests. The `Simulation` class has one method: `run_simulation`, that executes one run of our simulated Fab and returns the logged data from the run. For D.3, our `run_simulation` method supports arbitrary argument-passing to the `AMHS` to enable the implementation of adaptable vehicle routing policies.

4.4.8 Evaluation

We implement a function, `performance`, that calculates statistics to measure the performance of a vehicle routing policy. The `performance` function calculates the total number of movement requests completed during the recording period, the average latency of the completed movement requests, and the average throughput during the recording period. We ignore the burn-in period for the purpose of calculating these statistics. We focus on the average latency and throughput for comparing vehicle routing policies.

4.4.9 Animation

For our visualization per D.4, we implement a function to animate the results of a Fab simulation. Our `animate` function is built around the Python Matplotlib package, and more specifically, the `FuncAnimation` API [18]. The primary design constraint of our `animate` function is its speed. Therefore, we implement several optimizations geared towards performance engineering our software implementation. The `animate` function uses lookup tables to quickly access the graphical elements corresponding to each node and edge in the OHT system graph. We use the Matplotlib `blit` feature to avoid re-rendering the full image at each frame. Instead, we only re-render the graphical elements that are updated with each new frame.

The animation colors, size, and frame rate are all customizable. By default, our `animate` function produces videos that run at 5 frames per second and uses a green $\longrightarrow$ yellow $\longrightarrow$ red color scheme to represent the spectrum of unoccupied $\longrightarrow$ occupied regions of the OHT system graph. In this way, traffic jams may be quickly identified by the red regions of the animation. A snapshot from our `animate` function is depicted in Figure 4-3.

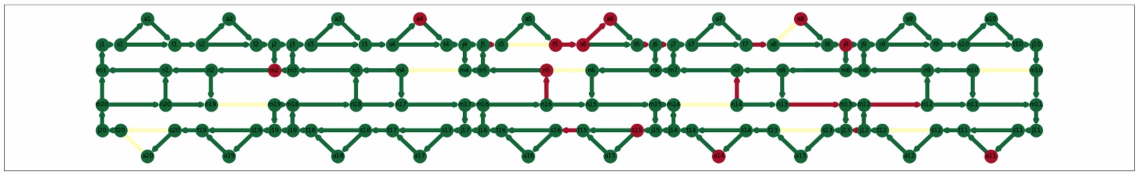

Figure 4-3: One frame from our animation of an example vehicle routing policy. Red represents at vehicle capacity, green represents no vehicles present, and yellow represents in between.

4.5 Summary Statistics

As a trial for our Fab simulation, we implement three vehicle routing policies and measure their performance on simulated runs. Using these vehicle routing policies, we are able to verify that our desiderata outlined in Section 4.1 are met.

We implement a subclass of `AMHS` called `StaticAMHS` that represents a vehicle routing policy parameterized by a shortest path search in a fixed, weighted graph. The `StaticAMHS` class precomputes and memorizes the results of the all pairs shortest path computation for the duration of the simulation. We add 1 to the weight of each edge to incorporate node travel times. This ensures that we route vehicles according to the true shortest path in the no-traffic setting.

We implement three subclasses of `StaticAMHS`, corresponding to three different policies for assigning vehicles to lot movement requests. These subclasses are called `GreedyAMHS`, `PayloadAMHS`, and `VehicleAMHS`. The `GreedyAMHS` vehicle routing policy assigns an arbitrary available vehicle to handle an arbitrary pending lot movement request as soon as both a lot and a vehicle are available. The `PayloadAMHS` vehicle routing policy considers the pending lot movement requests in an arbitrary order, then assigns the closest available vehicle to each lot movement request in sequence. The `VehicleAMHS` vehicle routing policy considers the available vehicles in an arbitrary order, then assigns each vehicle to the closest available lot with a pending movement request in sequence. All three vehicle routing policies use the precomputed shortest paths to route the vehicles through the OHT system graph.

We follow several more conventions from [19] during our simulated runs for D.9. We run simulations at four levels of increasing traffic: 37, 39, 41, and 43 vehicles.

Table 4.2: Results of simulation trial runs. The `GreedyAMHS` and `PayloadAMHS` policies achieve very poor performance regardless of the number of vehicles, whereas the `VehicleAMHS` policy exhibits significant performance degradation as the number of vehicles increases.

Routing Policy	Number of Vehicles	Latency	Throughput
`GreedyAMHS`	37	15402	15275
`GreedyAMHS`	39	15394	15977
`GreedyAMHS`	41	15810	16722
`GreedyAMHS`	43	15907	17400
`PayloadAMHS`	37	15476	15268
`PayloadAMHS`	39	15356	16017
`PayloadAMHS`	41	15775	16712
`PayloadAMHS`	43	15919	17473
`VehicleAMHS`	37	533	22007
`VehicleAMHS`	39	1061	23219
`VehicleAMHS`	41	1416	23729
`VehicleAMHS`	43	2706	24253

We scale up the rate of generated lot movement requests in these cases by 1.00, 1.05, 1.10, and 1.15, respectively. We use a burn-in period of 10 hours and a recording period of 24 hours. The relevant performance metrics are presented in Table 4.2.

Table 4.2 shows that the `GreedyAMHS` and `PayloadAMHS` policies are ineffective at efficiently routing vehicles through the OHT system graph. Both policies achieve 15000–16000 average latency of completed lot movement requests, regardless of the number of vehicles. We believe that 15000–16000 average latency corresponds to the maximum latency that can be realized under our simulation, with the vehicles moving at the slowest average rate that does not cause deadlock.

The `VehicleAMHS` policy is effective at the lowest level of vehicle traffic, but its performance degrades quickly as the number of vehicles increases beyond the critical threshold of 37. A 16% increase in the number of vehicles from 37 to 43 results in a >400% longer average latency.

We argue that this explosive performance degradation is a desirable feature of our simulation. This is because it is relatively easy to simulate no traffic at all or only standstill traffic. However, in order to derive useful training data from our simulated Fab, both extremes must be permissible under our simulation. According to queuing theory, physical systems tend to exhibit an exponential decrease in performance as the system approaches capacity. If our simulation emulates this behavior, then that is a good sign that we are achieving D.1. We hope to find that a small change to the number of vehicles or underlying routing policy should have a disproportionately large affect on the routing times. Table 4.2 shows that this is the case: the average latencies span multiple orders of magnitude. Our simulation is right on the edge between heavily congested traffic and manageable traffic, so that both extremes happen within a single run due to the randomness of our simulation.

The vehicle routing policies considered here still have a long way to go from the theoretical optimum. According to our specified lot movement request generation process, the lowest achievable average latency is 137 seconds: 18 seconds spent passing through nodes, 89 seconds spent passing through edges, 15 seconds spent acquiring the lot, and 15 seconds spent depositing the lot. So the best static vehicle routing policy among those considered is operating at 4× the best-case optimal average latency, which leaves significant room for improvement. Because the `VehicleAMHS` policy is most effective in all simulated runs, we use this policy to generate all of our training data in Section 4.6.

4.6 Data Generation

In order to extract useful training data from our Fab simulation, we must decide the following:

1. How to represent traffic data as a graph signal, which is a concept that we explain in Chapter 3, and
2. Which parameters to use for our selected vehicle routing policy.

Selecting a graph signal is a double-edged sword. More complex graph signals convey more information, but at the same time, more complex graph signals increase the difficulty of our prediction task. Therefore, we want to select graph signals that are both useful and possible to predict well. For simulated Fab traffic, we consider two candidate graph signals, which we call the snapshot signal and the compressed destination signal.

The snapshot graph signal consists of the tuple (number of vehicles, number of lots, edge weight w for the vehicle routing policy). This signal is taken at every node in our transformed OHT system graph. In the transformed graph, a node corresponds to a node or an edge in the original OHT system graph. The downside of the snapshot signal is that we are ignoring potentially valuable information about the eventual destination of each vehicle and lot. However, the snapshot signal has the advantage of being simple and interpretable. Also, the snapshot signal forces the supervised learning model to infer the distribution over vehicle and lot destinations. This has the potential benefit of giving us a built-in model for normal Fab operation.

The compressed destination graph signal consists of the tuple (number of vehicles separated by destination, number of lots separated by destination, edge weight w for the vehicle routing policy). The vehicles and lots are separated using a one-hot encoding for the destinations. Because we assume a low-dimensional graph signal, the components of the one-hot encoding correspond to the groups of nodes under our learned graph partition, which is described in Chapter 3. The compressed destination signal has the downside of being large and complex. Also, the compressed destination signal leads to a very sparse target signal that is more difficult to model. Since the input data depends on a learned parameter, we can encounter unstable, GAN-like performance with the possibility of mode collapse [36]. This signal has the advantage of using all the information that is available to us.

For our system, we select the snapshot graph signal. We believe its ease of use outweighs its downsides, and we want to avoid the difficulties of training a model that predicts the compressed destination signal. Also, we do not believe that the destination information merits usage because of its high space requirement.

To construct our graph signal, we also need to decide on an appropriate timescale. For reference, the authors of the original Graph WaveNet paper aggregate traffic data

Table 4.3: Numbers of time steps for different traffic datasets. The industry Fab dataset is the outlier, with a low number of time steps relative to the size of the graph.

Dataset	Nodes	Edges	Time Steps
METR-LA	207	1515	34272
PEMS-BAY	325	2369	52116
Industry Fab	13577	14403	27650
Simulated Fab	120	168	8640000
Transformed Simulated Fab	288	336	8640000

over 5-minute intervals and predict traffic 12 time steps into the future for a 1-hour prediction horizon [45]. Table 4.3 offers a comparison between different reference datasets and their total numbers of time steps.

Based on the results presented in Table 4.3, we decide to aggregate our data over 30-second intervals. To aggregate the traffic data, the snapshot graph signal becomes the average number of vehicles and the average number of lots over the aggregation window at each node in our transformed OHT system graph. With 30-second aggregation intervals, a prediction 12 time steps into the future gives us a 6-minute prediction horizon, which is a useful horizon for rerouting the vehicles in order to mitigate a pending traffic jam. Also, the signal aggregation has the benefit of making our graph signal less sparse and easier to predict. We are thus able to avoid using 1-second time steps, which would be too granular to be useful.

Next, we discuss which parameters we use for our selected vehicle routing policy. We generate training data according to 100 fixed vehicle routing policies. The policies were derived from shortest path searches on copies of the OHT system graph modified to have different edge weights. By fixed vehicle routing policy, we mean that the policy always selects the same route between any chosen starting and ending node, regardless of the presence of traffic in the simulated Fab. Fixed vehicle routing policies are theoretically less efficient than the alternative, which are referred to as dynamic vehicle routing policies, but they are useful for exploring the space of vehicle routing policies.

We design a statistical process for perturbing the edge weights controlling the shortest path search for our fixed vehicle routing policies. This statistical process corresponds to a hierarchical Bayesian model [12]. To guarantee a fixed vehicle routing policy, the parameters of this model are sampled once before the simulated run, and the results of the induced shortest path search are precomputed for runtime efficiency purposes.

The space of possible edge weights to control a shortest path search is very large. Therefore, we must select our generative model for edge weight perturbations carefully so that it explores a useful region of the vehicle routing policy space that is also as broad as possible. We engineer a perturbation model inspired by the changes a human might make to the edge weights in order to reroute traffic around an ongoing traffic jam. We introduce a notion of outlier nodes which are essential to our perturbation model. Outlier nodes have increased travel times so that the shortest path search will avoid these traffic choke points whenever possible. We use outlier nodes instead of outlier edges because we empirically observe a greater tendency for traffic to back up at the nodes rather than the edges.

In statistical notation, our generative model for edge weight perturbations is given by:

$$
\begin{aligned}
&\text{outlier\_prob} \sim \text{Uniform}(0.01, 0.25)\ , \\
&\text{for each node } i\ : \\
&\quad \text{is\_outlier}_i \sim \text{Bernoulli}(\text{outlier\_prob})\ , \\
&\quad \text{if is\_outlier}_i\ : \\
&\quad\quad w_i \leftarrow w_i + \text{Discrete\_Uniform}(1, 10)\ .
\end{aligned}
\tag{4.1}
$$

Our outlier_prob variable is a hyperparameter controlling the number of high-traffic outlier nodes. The Uniform distribution encodes an ignorance prior over the number of high-traffic outlier nodes. A value of 0.01 would correspond to almost no high-traffic outlier nodes, and a value of 0.25 would correspond to many high-traffic outlier nodes. Similarly, the Discrete_Uniform distribution over the perturbation magnitude encodes an ignorance prior over the degree of traffic present in each high-traffic outlier node. A value of 1 would correspond to almost no traffic at the node, and a value of 10 would correspond to heavily congested traffic at the node. Our broad prior distribution allows for many possible edge weight perturbations. Note that our generative model for edge weight perturbations only ever increases the edge weights and never decreases them because congestion can slow down the vehicles in our simulated Fab but never speed them up.

Tables 4.4 to 4.7 catalog the results of 100 simulated training data runs. Instead of sampling the outlier probability from our specified $\text{Uniform}(0.01, 0.25)$ prior distribution, we performed runs with a deterministic sequence of outlier probabilities: 0.01, 0.02, 0.03, ..., 0.24, 0.25. This was to reduce the variance of our runs and guarantee training data exposure to as broad a region of our prior distribution as possible. The is_outlier indicator variables and edge weight perturbations were still randomly sampled as specified in Equation 4.1.

Tables 4.4 to 4.7 highlight both sides of the critical threshold for the number of vehicles. Above this threshold, latency grows enormously as vehicles are added to the simulated Fab. In Table 4.4, the observed latencies vary between 518 and 1096, with of the most observed latencies <1000. In contrast, Table 4.7 has minimum observed latency 1559 and maximum observed latency 3818. Tables 4.5 and 4.6 are intermediaries along the transition to higher latency and variance of the latency observations.

4.7 Containerization

Over the course of this project, we consulted with Samsung and worked with their proprietary simulator. To facilitate interaction with the Samsung proprietary software, we create a containerization environment using the Docker project [31]. Docker is a lightweight virtual machine replacement which allows for consistent runtime conditions across multiple systems. The Docker containerization environment allows the user to more easily control the parameters of the Samsung proprietary software.

Table 4.4: Results of the generated training data runs with 37 vehicles. Regardless of the outlier probability, all the randomly sampled routing policies are reasonably effective at completing the lot movement requests with low latency.

Number of Vehicles	Outlier Prob	Latency	Throughput
37	0.01	670	21943
37	0.02	518	21905
37	0.03	694	21935
37	0.04	701	21726
37	0.05	686	21937
37	0.06	756	21993
37	0.07	725	21945
37	0.08	586	21911
37	0.09	596	21713
37	0.10	683	21918
37	0.11	819	21750
37	0.12	660	22218
37	0.13	1096	22143
37	0.14	982	21921
37	0.15	587	21805
37	0.16	608	21958
37	0.17	752	21915
37	0.18	734	21856
37	0.19	781	21754
37	0.20	849	21942
37	0.21	1024	21996
37	0.22	768	22045
37	0.23	718	22099
37	0.24	637	21966
37	0.25	957	22165

Table 4.5: Results of the generated training data runs with 39 vehicles. We are beginning to see higher variance in the results, with certain simulated runs struggling to maintain low latency.

Number of Vehicles	Outlier Prob	Latency	Throughput
39	0.01	1030	22981
39	0.02	1532	22789
39	0.03	1080	22946
39	0.04	1655	22710
39	0.05	1025	22981
39	0.06	995	22959
39	0.07	1499	22713
39	0.08	1568	22840
39	0.09	902	22821
39	0.10	912	22993
39	0.11	888	22968
39	0.12	1271	23033
39	0.13	1241	22564
39	0.14	969	22876
39	0.15	1006	22851
39	0.16	1848	22731
39	0.17	877	22917
39	0.18	930	22946
39	0.19	1197	22855
39	0.20	1268	22875
39	0.21	1274	22919
39	0.22	1454	22793
39	0.23	1608	22843
39	0.24	1392	22754
39	0.25	1530	22722

Table 4.6: Results of the generated training data runs with 41 vehicles. Nearly all the simulated runs have greatly slowed down relative to the 37-vehicle runs. The overall Fab throughput remains relatively constant regardless of the outlier probability.

Number of Vehicles	Outlier Prob	Latency	Throughput
41	0.01	1926	23489
41	0.02	1793	23824
41	0.03	1928	23488
41	0.04	1149	23537
41	0.05	1710	23931
41	0.06	2117	23340
41	0.07	1956	23529
41	0.08	2339	23714
41	0.09	2087	23815
41	0.10	1773	23645
41	0.11	1452	23588
41	0.12	1948	23313
41	0.13	1715	23671
41	0.14	1902	23475
41	0.15	1454	23771
41	0.16	2384	23702
41	0.17	2172	23388
41	0.18	1751	23822
41	0.19	2168	23257
41	0.20	2128	23899
41	0.21	1855	23652
41	0.22	2064	23785
41	0.23	1122	23707
41	0.24	1643	23648
41	0.25	1152	23801

Table 4.7: Results of the generated training data runs with 43 vehicles. Representing the worst results out of all our simulated runs, the 43-vehicle runs see an explosive increase in the average latency over the recording period.

Number of Vehicles	Outlier Prob	Latency	Throughput
43	0.01	2702	24033
43	0.02	2155	24473
43	0.03	2112	24151
43	0.04	2087	23967
43	0.05	2484	24458
43	0.06	2866	24027
43	0.07	1559	24683
43	0.08	3085	24508
43	0.09	2060	24203
43	0.10	2831	24185
43	0.11	3324	23949
43	0.12	2236	24567
43	0.13	2227	24269
43	0.14	2691	24085
43	0.15	3818	23865
43	0.16	2263	24399
43	0.17	2528	24232
43	0.18	1977	24411
43	0.19	3451	23975
43	0.20	2968	24121
43	0.21	3322	23797
43	0.22	3388	23685
43	0.23	2159	24346
43	0.24	2231	24184
43	0.25	2419	24002

Chapter 5

Experiments

The goal of this chapter is to compare Partition WaveNet to alternative approaches for spatiotemporal modeling on large graphs. We first describe our constructed spatiotemporal datasets, picking up where we left off in Chapter 4. After, we explain our benchmark models and our process for designing these benchmarks. The purpose of the benchmarks is to measure the performance of different methods for reducing the size of the OHT system graph. Then, we go over our computational setup and present our experimental results, focusing on our inferences about why the models achieve their measured performance. We also identify simulations and analyses for future exploration of the performance of Partition WaveNet. We conclude with a discussion about potentially interesting follow-up research projects.

5.1 Constructed Datasets

Our codebase is built on an improved Graph WaveNet implementation [40]. In addition to using better hyperparameters, the authors add skip connections and learning rate decay to improve the performance of Graph WaveNet on the reference datasets. We do not change the missing data representation as described in [40] because our snapshot graph signal does not represent traffic speed, but instead represents the rate of congestion as measured by the average number of vehicles. The improved Graph WaveNet implementation has convenient helper functions for:

1. Correctly constructing the Graph WaveNet adjacency matrix from a list of edges (`gen_adj_matrix.py`),
2. Generating the training, validation, and testing datasets by using a moving window over the time dimension (`generate_training_data.py`), and
3. Summarizing the performance of trained models with a concise set of metrics (`exp_results.py`).

We modify the dataset generation tool in `generate_training_data.py` to add another feature to the constructed datasets: the edge weight w for the vehicle routing policy. The original, unmodified dataset generation tool only supports one, temporally contiguous traffic run as the input data. We generalize the implementation to support an arbitrary number of temporally contiguous traffic runs as the input data. This change allows us to create new datasets from our 100 simulated training data runs summarized in Tables 4.4–4.7.

The three model preparation tools require particular formatting of the inputs in order to function properly. Our simulated Fab model from Chapter 4 outputs data that does not match the formatting needs of the model preparation tools. Therefore, we implement a set of preprocessing tools, included in our codebase as `AMHS_preprocess.ipynb`, to serve as an intermediary between our simulated Fab model and the model preparation tools. `AMHS_preprocess.ipynb` aggregates graph signals at the desired intervals, performs shortest path searches and graph partitions, and reformats the data to meet the needs of the model preparation tools.

The process for constructing the Graph WaveNet adjacency matrix is to take the shortest path length between all pairs of nodes in the OHT system graph, transformed first by Z-score normalization and then by a thresholded Gaussian kernel [45]. Mathematically, this construction process is given by

$$A_{ij} = \max\left(e^{-\frac{\mathrm{spl}_{ij}^2}{\sigma}} - k,\ 0\right) \ , \tag{5.1}$$

where spl_{ij} represents the shortest path length between nodes i and j, σ represents the standard deviation of all the spl_{ij} values, and k is some suitably chosen constant. Following the literature, we use $k = 0.1$. We follow the convention that $\text{spl}_{ii} \neq 0$, but instead equals the shortest path length of the self-loop in the OHT system graph from node i to itself. This convention allows the adjacency matrix to better capture the effect of a node on the future state of that node. For example, vehicles caught in traffic at a node may follow a loop in the OHT system graph and return to the same node at a future point in time.

The constructed datasets are separated chronologically with 70% for training, 10% for validation, and 20% for testing. In the case of multiple input traffic runs, we separate each run chronologically. This guarantees that each run is represented in the training, validation, and testing datasets.

In total, we construct four AMHS traffic datasets. These datasets consist of two data sources crossed with two graph formats. Our data sources are:

1. The data source of 100 simulated training data runs generated according to differently perturbed, fixed vehicle routing policies (FAB).

2. A toy data source of four simulated training data runs, one with each of 37, 39, 41, and 43 vehicles, generated according to the unperturbed, fixed vehicle routing policy (TOY).

The purpose of the TOY data source is to provide a sanity checking problem that we expect to be able to model well. The TOY data source leads to datasets that give faster iteration on model training so that we can more quickly debug and get feedback on our trained models. Our two graph formats are:

1. The transformed OHT system graph, in which a node corresponds to a node or an edge in the original OHT system graph (FULL).

2. The transformed OHT system graph with a hand-selected graph partition applied to reduce the size of the graph (PARTITION).

For the purpose of calculating the adjacency matrix, we must assign distances under our hand-selected graph partition. In the PARTITION graph format, we set the

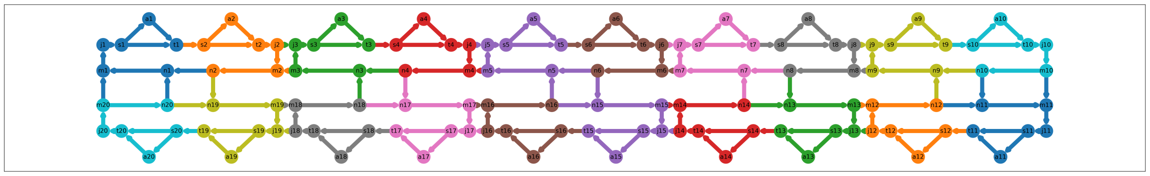
Figure 5-1: The original OHT system graph colored according to our hand-selected graph partition. The duplicated colors on opposite diagonals of the graph correspond to different supernodes under the partition.

distance between a pair of supernodes equal to the average distance from a node in the first supernode to a node in the second supernode. For the edge weight w feature of the snapshot graph signal, we use the average perturbed edge weight in the supernode. The graph signals are aggregated at each supernode under the hand-selected graph partition. All of the partition modifications are carried out in `AMHS_preprocess.ipynb`.

We only use the PARTITION graph format for our partition benchmark model, which we describe in greater detail in Section 5.2. Our hand-selected graph partition is depicted in Figure 5-1. The partition has one supernode for every tool in the simulated Fab. Each node is grouped with the closest tool, and each directed edge (u, v) is grouped with node v.

We present summary statistics for our four constructed datasets in Table 5.1. To reduce the storage burden of our datasets, we perform several reductions to simplify our problem. These reductions are undesirable because they force our models to specialize to problems that are more narrow in scope. However, we find the reductions necessary to build tractable datasets. We discuss the unreduced task further in Section 5.5.

For our first reduction, instead of modeling nodes and edges in the original OHT system graph, we only model the edges in the original OHT system graph. Vehicles at nodes are treated as though they still have not left the previous edge. Second, instead of predicting the three entries of the snapshot graph signal, we only predict two, leaving out the number of lots feature. This restricts the scope of our prediction problem to vehicle traffic instead of vehicle traffic and lot movement requests. Third,

Table 5.1: The shapes of our four constructed datasets. The FAB data source leads to significantly larger datasets because it is based on the results of 100 simulated runs instead of only four.

Data Source	Graph Format	Time Steps	Window	Nodes	Features
FAB	FULL	47700	12	168	2
FAB	PARTITION	47700	12	20	2
TOY	FULL	5716	12	168	2
TOY	PARTITION	5716	12	20	2

we add a stride to our time dimension, analogous to the stride used in neural network convolutions. We only create data points from time steps at a specified period. We use a stride of two for the TOY data source and a stride of six for the FAB data source. These simplifications reduce the size of our largest dataset, FAB FULL, by ~93.5%. Even so, FAB FULL takes >256 MB to store on disk.

5.2 Benchmark Models

The purpose of our benchmark models is to provide a concise set of alternatives for comparison to Partition WaveNet. Because the computational cost of training another model is high, we strive to select a minimal set of benchmarks that would still allow us to draw conclusions about the performance of Partition WaveNet. Therefore, each benchmark model is carefully selected to provide the most useful comparisons, leading us to exclusively use deep learning models as benchmarks. Since we expect non-deep learning methods to underperform relative to Partition WaveNet, the non-deep learning methods would be less useful as benchmarks.

5.2.1 Selected Benchmark Models

We use a suite of three deep learning models as benchmarks:

1. Graph WaveNet.

2. Graph WaveNet without a prespecified adjacency matrix, and with added linear

input and output layers to reduce the dimensionality of the problem. Since there is no prespecified adjacency matrix, Graph WaveNet uses only the self-adaptive adjacency matrix. We refer to this model as the embedding benchmark model.

3. Graph WaveNet applied to the PARTITION graph format. Graph WaveNet uses the self-adaptive adjacency matrix in addition to the PARTITION adjacency matrix. We refer to this model as the partition benchmark model.

We have different expectations about how each of these models would perform relative to Partition WaveNet. Our expectations dictate the intended purpose of each benchmark model.

We expect the Graph WaveNet model to outperform Partition WaveNet with respect to MAE because the latter equals the former with an added handicap of the normalized node embedding. We purposefully construct our datasets so that Graph WaveNet is tractable, even though the intended purpose of Partition WaveNet is for settings in which Graph WaveNet is infeasible to train. By using Graph WaveNet as a benchmark, we can measure the performance degradation imposed by the normalized node embedding. Therefore, Graph WaveNet serves as an upper bound on the performance of Partition WaveNet.

We expect Partition WaveNet to outperform the embedding benchmark model with respect to MAE. The embedding benchmark model is analogous to a state-of-the-art temporal network with an end-to-end network embedding applied to its spatial inputs. The adjacency structure from the original graph is lost with the embedding, so we cannot specify an adjacency matrix using the embedding benchmark model. We thus expect Partition WaveNet to outperform the embedding benchmark model because the embedding benchmark model does not fully leverage the known spatial information encoded in the adjacency matrix. The embedding benchmark model helps us quantify the predictive value of the known spatial information.

We also expect Partition WaveNet to outperform the partition benchmark model with respect to MAE. The partition benchmark model is a multi-stage model. First, the state-of-the-art Graph WaveNet model is applied to a hand-reduced graph. This model predicts the average traffic in each supernode under the selected partition. Then, the predicted average traffic in each supernode is used to predict the average traffic in each constituent node. Constituent node traffic is predicted with a linear model. We use the linear regression implemented by the Python statsmodels package [39].

We expect Partition WaveNet to outperform the partition benchmark model because Partition WaveNet automatically selects a graph partition to aid in the prediction task. We expect an automatically selected graph partition to be more suitable than a hand-selected graph partition. The partition benchmark model thus helps us measure the predictive value of the automatic partition selection.

The partition and embedding benchmark models use the same reduced dimension as the b parameter defined in Section 3.2. This guarantees that we are comparing networks with approximately the same numbers of parameters. Therefore, each of our alternative models performs the same degree of dimensionality reduction as does Partition WaveNet.

5.2.2 Rejected Benchmark Models

In this section, we consider several potential models that we do not select as benchmark models. We describe these alternative benchmark models and our rationale for rejecting them.

We consider three alternative benchmark models below:

1. Graph GRU, an RNN-based deep spatiotemporal network architecture that is suitable for large graphs [49].

2. WaveNet with a network embedding applied to the inputs.

3. The embedding benchmark model with the PARTITION adjacency matrix.

Our issue with Graph GRU as a benchmark is that the superiority of CNN-based temporal models, and specifically the WaveNet dilated causal convolutions, has been demonstrated in [34, 45]. We therefore expect Graph GRU to perform worse than Graph WaveNet and to have a higher computational cost of training.

We reject WaveNet as a benchmark because Graph WaveNet without a specified adjacency matrix is a superior benchmark due to its self-adaptive adjacency matrix. Therefore, we prefer our embedding benchmark model.

We consider the embedding benchmark model with the PARTITION adjacency matrix to be less suitable than our selected benchmarks because we believe it does not constitute a principled usage of the PARTITION adjacency matrix. We prefer to create alternative datasets using the PARTITION graph format and train the unmodified Graph WaveNet architecture on these alternative datasets, as we do for our partition benchmark model.

5.3 Computational Setup

We conduct our experiments under various computer environments provided by Amazon Web Services (AWS), a popular cloud computing platform. AWS provides on-demand access to virtual computers called instances, which can emulate hardware CPUs and GPUs for processing.

Our simulated training data runs for the FAB and TOY data sources are carried out by one `t2.2xlarge` Amazon Elastic Compute Cloud instance. This instance type guarantees eight virtual CPU cores and 32 GB of RAM. Our instance is attached to one 16 GB Amazon Elastic Block Storage volume to store the generated data. We perform our simulated training data runs in parallel, so that one run occurs on each of the eight virtual CPU cores simultaneously.

To speed up training on the FAB data source, we leverage GPU compute resources, which are the fastest hardware components available for training neural networks. Our GPU computing environment consists of one `g4dn.xlarge` Amazon Elastic Compute Cloud instance. This instance type guarantees four virtual CPU cores, 16 GB of RAM, and 125 GB of instance storage. The `g4dn.xlarge` instance also guarantees one NVIDIA T4 Tensor Core GPU with 16 GB of GPU memory.

The `t2.2xlarge` instance is used to train all our models on the TOY data source. These models are trained on CPU by setting the PyTorch device to `"cpu"` instead of `"cuda:0"`. The `g4dn.xlarge` instance is used to train all our models on the FAB data source. These models are trained on GPU by setting the PyTorch device to `"cuda:0"`.

5.4 Experimental Results

In total, we train eight models: Partition WaveNet plus three benchmark models crossed with two data sources. In this section, we compare the training and performance of Partition WaveNet to that of the other models. Our general findings are that Partition WaveNet trains nearly as quickly as the partition and embedding benchmark models, and outperforms both with respect to MAE.

Our model training times are presented in Table 5.2. Graph WaveNet takes significantly longer to train than the other three models on both data sources. This is because Partition WaveNet and our partition and embedding benchmark models correspond to Graph WaveNet on a 20-node graph; the full Graph WaveNet has a 168-node graph, and it requires much more training time as a result. Among the other three models, Partition WaveNet is the slowest by a small margin. This is the expected result because of the training cost associated with training the normalized node embedding.

The recorded model training times are skewed on the TOY data source because the embedding and partition benchmark models are not trained for the full 100 epochs. The early stopping feature of the Graph WaveNet training policy halts model training if no improvement to validation loss is observed for 20 epochs. Therefore, our embedding benchmark model trains for 46 epochs, and our partition benchmark model trains for 35 epochs. We suspect that the early stopping is invoked because the TOY data source has too few time steps, which inhibits the embedding and partition benchmark models from generalizing well. Early stopping does not occur on the more broad FAB data source.

Table 5.2: The training times of our eight models. Although the FAB data source is ∼10× larger than the TOY data source, the FAB data source models were trained on a GPU, which leads to much shorter training times.

Model	Data Source	Hours	Minutes
Partition WaveNet	TOY	4	2.68
Graph WaveNet	TOY	30	39.13
Embedding Benchmark	TOY	1	57.46
Partition Benchmark	TOY	0	55.09
Partition WaveNet	FAB	0	44.70
Graph WaveNet	FAB	3	59.12
Embedding Benchmark	FAB	0	38.06
Partition Benchmark	FAB	0	39.82

Our measured training time for Graph WaveNet is consistent with the literature. Our FAB FULL dataset is approximately the same size as the METR-LA dataset used in [40]. The authors report four hours of training time is needed to train Graph WaveNet on a dataset of this size on an NVIDIA T4 Tensor Core GPU. Our Graph WaveNet model trains in almost exactly four hours on the same hardware.

We present the performance of our eight models in Table 5.3. On both data sources, Graph WaveNet is the best-performing model by a wide margin. This suggests that modeling the full graph yields a large benefit to the performance of the resulting model. This result is expected given the state-of-the-art status of Graph WaveNet. On both data sources, Partition WaveNet is the runner-up to Graph WaveNet. Partition WaveNet achieves a testing MAE of 0.185 on the TOY data source and 0.183 on the FAB data source. In comparison, Graph WaveNet achieves a testing MAE of 0.164 on the TOY data source and 0.163 on the FAB data source. The magnitude of the observed MAEs is approximately the same for both data sources.

Our validation MAE column in Table 5.3 suggests that no model is substantially overfitting to the data. Our testing MAE column suggests that there is no measurable difference between the training data distribution and the testing data distribution. Therefore, we believe that we generated enough data for the training, validation, and testing datasets, to all be representative of the typical traffic in our simulated Fab.

Table 5.3: Model training, validation, and testing dataset performance. The loss function ℓ is the mean MAE across the length-12 prediction window, and the Epoch column records the index of the best epoch from 1 to 100. We report two performance measurements for our partition benchmark model: the performance of the model on the PARTITION data format, and the performance of the derived linear model that predicts the FULL data format.

Model	Data Source	Train ℓ	Valid ℓ	Test ℓ	Epoch
Partition WaveNet	TOY	0.183	0.184	0.185	97
Graph WaveNet	TOY	0.163	0.164	0.164	93
Embedding Benchmark	TOY	0.199	0.199	0.200	26
Partition Benchmark	TOY	0.831	0.841	0.845	15
P. B. Linear Output	TOY	0.215	0.207	0.213	—
Partition WaveNet	FAB	0.184	0.182	0.183	98
Graph WaveNet	FAB	0.165	0.162	0.163	98
Embedding Benchmark	FAB	0.195	0.193	0.195	93
Partition Benchmark	FAB	0.834	0.830	0.830	96
P. B. Linear Output	FAB	0.193	0.191	0.187	—

This is important to verify because our datasets are separated chronologically, so we do not want a later time to correspond to a different traffic regime.

The embedding and partition benchmark models perform better on the FAB data source than on the TOY data source. For Partition WaveNet and Graph WaveNet, there is little difference in performance between the two data sources. The underperforming embedding and partition benchmark models correspond to the models that exhibit early stopping during training. We suspect that the early stopping is not the cause of the underperformance, but is rather a symptom. The embedding and partition benchmark models do not generalize well on the TOY data source, so the training process is not able to improve either model beyond a certain epoch.

It is interesting to note the relative difficulty of the intermediate prediction task on the PARTITION data format, which corresponds to the partition benchmark model in Table 5.3. We might expect the PARTITION data format to correspond to an easier task than the FULL data format because there are fewer nodes to predict, as the graph signals are aggregated at the supernodes. However, the smaller number

Table 5.4: Model testing dataset performance at four different prediction horizons. The $t = 30$s column corresponds to the first prediction horizon, and the $t = 360$s column corresponds to the last prediction horizon. We report MAE loss for all models.

Model	Data Source	$t = 30$s	$t = 90$s	$t = 180$s	$t = 360$s
Partition WaveNet	TOY	0.169	0.179	0.186	0.191
Graph WaveNet	TOY	0.124	0.150	0.169	0.175
Embedding Benchmark	TOY	0.200	0.200	0.199	0.200
Partition Benchmark	TOY	0.608	0.808	0.875	0.917
P. B. Linear Output	TOY	0.211	0.209	0.219	0.212
Partition WaveNet	FAB	0.156	0.177	0.186	0.192
Graph WaveNet	FAB	0.115	0.157	0.169	0.175
Embedding Benchmark	FAB	0.195	0.195	0.195	0.195
Partition Benchmark	FAB	0.583	0.780	0.861	0.907
P. B. Linear Output	FAB	0.172	0.174	0.192	0.200

of nodes means that the partition benchmark model has a smaller sized input as well. We believe that the poor performance of the partition benchmark model before the linear output is caused by the applied graph partition obfuscating the original problem.

We present the performance of our eight models as a function of the prediction horizon in Table 5.4. We observe lower MAE at the shorter prediction horizons and higher MAE at the longer prediction horizons for most models. This trend is broken by the partition benchmark linear output on the TOY data source, and by the embedding benchmark model on both data sources. We hypothesize that the embedding benchmark model predicts uniformly poorly across prediction horizons because of the presence of two linear layers. The embedding benchmark model uses a linear layer to scale up the inputs to Graph WaveNet and a linear layer to scale up the outputs from Graph WaveNet. We suspect that the interaction between the two linear layers and the Graph WaveNet architecture causes this undesirable behavior.

Once again, Graph WaveNet is the best in terms of MAE. Graph WaveNet performs particularly well on the shortest prediction horizon $t = 30$s. Partition WaveNet lags behind Graph WaveNet in terms of MAE, but outperforms the embedding and partition benchmark models on both data sources. Partition WaveNet is the only reduced benchmark to consistently exhibit the expected behavior across different prediction horizons.

5.5 Future Analyses

There are several additional simulations that would be helpful to perform in order to better understand the performance of Partition WaveNet as it relates to the AMHS traffic control problem. First, it would be useful to run Partition WaveNet on the unreduced versions of our constructed datasets. The unreduced datasets would have 288 nodes instead of 168 nodes and would have three features instead of two features. The stride reduction would still be needed to store our datasets. Training models on the unreduced datasets could take $>2\times$ the amount of training time, depending on the caching behavior of the hardware. This change would give us insight into the full AMHS modeling problem that we are trying to solve.

It would be useful to simulate Partition WaveNet with many different b parameters to get a better sense of the tradeoff between spatial compression, training time, and performance. We would start by approximating the curve that relates the training time to the performance of Partition WaveNet. This curve would be analogous to an ROC curve, and points along the curve would be parameterized by the b parameter of Partition WaveNet, i.e., its level of spatial compression.

Another suitable simulation to run would be training Partition WaveNet on a sequence of graphs, sized in increasing orders of magnitude. This experiment would help us measure the scalability of Partition WaveNet in the extremely large graph regime. A corresponding real-world experiment that would give us insight into the scalability of Partition WaveNet would be to train Partition WaveNet on data from a large semiconductor Fab. Because a large semiconductor Fab is the intended setting for Partition WaveNet, this experiment would give the most relevant measure of the performance of Partition WaveNet.

5.6 Further Directions

In Section 5.4, we find that Partition WaveNet outperforms the embedding and partition benchmark models on both data sources. Even so, there may be more successful extensions to Partition WaveNet based on similar ideas and architectures. First, it would be interesting to see how different structural modifications affect the performance of Partition WaveNet. For example, we could replace our node embedding dictionary $\mathbf{B}_{emb}$ with a source and a destination node embedding dictionary. Under this extension, we lose the physical interpretation of the node embedding dictionary, but it would be interesting to observe the performance of the resulting model.

In this thesis, we only scratch the surface of AMHS modeling tasks. We do not explore the process of deriving vehicle routing policies from a learned Partition WaveNet traffic model in depth. It would be interesting to compare the performance of various tree search algorithms on the task of deriving vehicle routing policies. There are many unanswered questions in this domain, e.g., whether breadth-first (alpha-beta pruning) or depth-first (Monte Carlo Tree Search) is a more effective search strategy, or, whether it is better to update a dynamic policy frequently based on short searches or infrequently based on long searches. Our simulated Fab could be extended with a dynamic vehicle routing policy class parameterized by an arbitrary traffic prediction function so that models can be easily compared on the basis of their performance as a vehicle routing policy.

We also do not explore the specifics of anomaly detection based on a learned Partition WaveNet traffic model in much detail. Partition WaveNet gives us the flexibility to frame the anomaly detection problem as supervised learning if we treat anomalies as labeled graph signals, as unsupervised learning if we cluster predicted outputs, or as semi-supervised learning if we take an in-between approach. In the unsupervised learning case, it would be interesting to compare heuristic approaches to established machine learning algorithms for clustering.

Given the success of Partition WaveNet in the AMHS modeling domain, it is natural to ask if Partition WaveNet would be equally successful in other domains

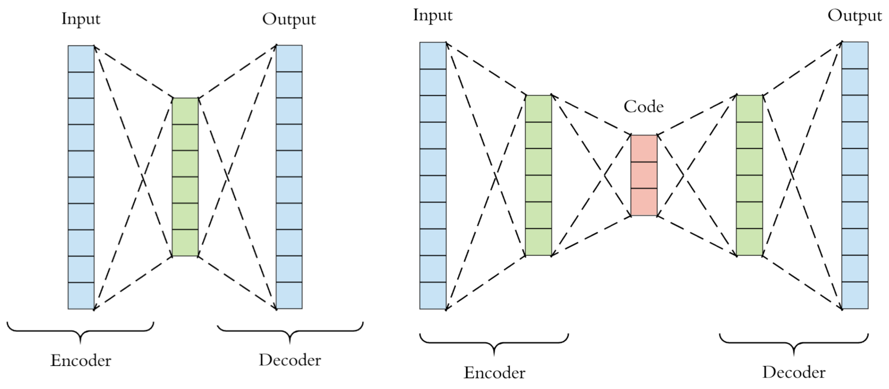

Figure 5-2: A schematic of deep autoencoder architectures with less restricted and more restricted induced graph representations.

that deal with large system graphs. For example, Partition WaveNet could be used to model temporal social network signals or the spread of disease via a contact graph. Graphs corresponding to social interactions can have millions of nodes and edges, so it would be interesting to measure the performance of Partition WaveNet in the extremely large graph regime.

Also, it is worth investigating whether deep variational methods for learning graph embeddings are successful at tasks within the AMHS domain. Deep variational graph autoencoders inspired the partition encoding architecture of Partition WaveNet, so if this framework can succeed at other AMHS-related tasks, then it would support the success of Partition WaveNet at general traffic modeling in AMHS. It would be interesting to compare degradation in the performances of Partition WaveNet and of deep variational graph autoencoders as a function of the induced graph representation size b. In Figure 5-2, the more restricted induced graph representation labeled Code has $b = 3$ because we force the autoencoder to compress the signal into a three-dimensional latent space.

It would be interesting to apply Partition WaveNet directly to the dynamic traffic routing task rather than through general-purpose traffic modeling as an intermediary. To do this, training examples would be constructed in real-time using simulations, and Partition WaveNet would attempt to optimize a heuristic objective function corresponding to traffic management using graph signals consisting of both current network traffic and tunable parameters to control the AMHS routing protocol. This experiment would border on being computationally infeasible due to its similarities with deep reinforcement learning methods, but it could also help understand the behavior of Partition WaveNet on a deeper level.

Bibliography

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

[2] Gowtham Atluri, Anuj Karpatne, and Vipin Kumar. Spatio-temporal data mining: A survey of problems and methods. *ACM Computing Surveys (CSUR)*, 51(4):83, 2018.

[3] Kelly Bartlett, Junho Lee, Shabbir Ahmed, George Nemhauser, Joel Sokol, and Byungsoo Na. Congestion-aware dynamic routing in automated material handling systems. *Computers & Industrial Engineering*, 70:176–182, 2014.

[4] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[5] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[6] Victor Castillo. Parallel simulations of manufacturing processing using SimPy, a python-based discrete event simulation tool. In *Proceedings of the 2006 Winter Simulation Conference*, pages 2294–2294. IEEE, 2006.

[7] Jianping Chen, Beixin Xia, and Xin Chen. Effectiveness of vehicle dynamic reassignment dispatching in interbay material handling system. In *2016 International Conference on Artificial Intelligence and Engineering Applications*. Atlantis Press, 2016.

[8] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):833–852, 2018.

[9] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org, 2017.

[10] Mohamed Khalil El Mahrsi and Fabrice Rossi. Graph-based approaches to clustering network-constrained trajectory data. In *International Workshop on New Frontiers in Mining Complex Patterns*, pages 124–137. Springer, 2012.

[11] Fabio Galasso, Margret Keuper, Thomas Brox, and Bernt Schiele. Spectral graph reduction for efficient image and streaming video segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 49–56, 2014.

[12] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. CRC Press, 2013.

[13] Aric Hagberg, Pieter Swart, and Daniel S. Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Laboratory.(LANL), Los Alamos, NM (United States), 2008.

[14] Binh Han, Ling Liu, and Edward Omiecinski. NEAT: Road network aware trajectory clustering. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 142–151. IEEE, 2012.

[15] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[16] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[17] Renjun Hu, Charu C. Aggarwal, Shuai Ma, and Jinpeng Huai. An embedding approach to anomaly detection. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 385–396. IEEE, 2016.

[18] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[19] Illhoe Hwang and Young Jae Jang. Q (λ) learning-based dynamic route guidance algorithm for overhead hoist transport systems in semiconductor fabs. *International Journal of Production Research*, 58(4):1–23, 2019.

[20] Ashesh Jain, Amir R. Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-RNN: Deep learning on spatio-temporal graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5308–5317, 2016.

[21] Younkook Kang, Sungwon Lyu, Jeeyung Kim, Bongjoon Park, and Sungzoon Cho. Dynamic vehicle traffic control using deep reinforcement learning in automated material handling system. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9949–9950, 2019.

[22] Ahmed Kharrat, Iulian Sandu Popa, Karine Zeitouni, and Sami Faiz. Clustering algorithm for network constraint trajectories. In *Headway in Spatial Data Handling*, pages 631–647. Springer, 2008.

[23] Haejoong Kim and Dae-Eun Lim. Deep-learning-based storage-allocation approach to improve the AMHS throughput capacity in a semiconductor fabrication facility. In *Asian Simulation Conference*, pages 232–240. Springer, 2018.

[24] Haejoong Kim, Dae-Eun Lim, and Sangmin Lee. Deep learning-based dynamic scheduling for semiconductor manufacturing with high uncertainty of automated material handling system capability. *IEEE Transactions on Semiconductor Manufacturing*, 33(1):13–22, 2020.

[25] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[26] Thomas N. Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[27] Hiroshi Kondo. Advanced simulation framework for AMHS. In *2007 International Symposium on Semiconductor Manufacturing*, pages 1–4. IEEE, 2007.

[28] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

[29] Renjie Liao, Marc Brockschmidt, Daniel Tarlow, Alexander L Gaunt, Raquel Urtasun, and Richard Zemel. Graph partition neural networks for semi-supervised classification. *arXiv preprint arXiv:1803.06272*, 2018.

[30] James T. Lin, Fu-Kwun Wang, and Chun-Kuan Wu. Simulation analysis of the connecting transport AMHS in a wafer fab. *IEEE Transactions on Semiconductor Manufacturing*, 16(3):555–564, 2003.

[31] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014.

[32] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. Gap: Generalizable approximate graph partitioning framework. *arXiv preprint arXiv:1903.00614*, 2019.

[33] Yoshio Nishi and Robert Doering. *Handbook of Semiconductor Manufacturing Technology*. CRC Press, 2007.

[34] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[35] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder for graph embedding. *arXiv preprint arXiv:1802.04407*, 2018.

[36] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[37] Yukio Sadahiro, Raymond Lay, and Tetsuo Kobayashi. Trajectories of moving objects on a network: detection of similarities, visualization of relations, and classification of trajectories. *Transactions in Geographic Information Systems*, 17(1):18–40, 2013.

[38] Robert Schmaler, Christian Hammel, Thorsten Schmidt, Matthias Schoeps, Joerg Luebke, and Ralf Hupfer. Strategies to empower existing automated material handling systems to rising requirements. *IEEE Transactions on Semiconductor Manufacturing*, 30(4):440–447, 2017.

[39] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*, volume 57, page 61. Scipy, 2010.

[40] Sam Shleifer, Clara McCreery, and Vamsi Chitters. Incrementally improving Graph WaveNet performance on traffic prediction. *arXiv preprint arXiv:1912.07390*, 2019.

[41] Gian Antonio Susto, Matteo Terzi, and Alessandro Beghi. Anomaly detection approaches for semiconductor manufacturing. *Procedia Manufacturing*, 11:2018–2024, 2017.

[42] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913. IEEE, 2012.

[43] Senzhang Wang, Jiannong Cao, and Philip S Yu. Deep learning for spatio-temporal data mining: A survey. *arXiv preprint arXiv:1906.04928*, 2019.

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[45] Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, and Chengqi Zhang. Graph WaveNet for deep spatial-temporal graph modeling. *arXiv preprint arXiv:1906.00121*, 2019.

[46] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875*, 2017.

[47] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[48] Wenchao Yu, Cheng Zheng, Wei Cheng, Charu C. Aggarwal, Dongjin Song, Bo Zong, Haifeng Chen, and Wei Wang. Learning deep network representations with adversarially regularized autoencoders. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2663–2671. ACM, 2018.

[49] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. GAAN: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.