Learning to Dispatch: A Reinforcement Learning Framework for Train Dispatch Networks

Andres Espinosa

Industrial and Systems Engineering
University of Florida
andresespinosa@ufl.edu

Abstract.

1 Introduction

The Train Dispatching Problem (TDP) concerns the real-time management of train movements across a rail network, ensuring safe and efficient use of infrastructure. This involves deciding when and where trains should move, stop, or wait, based on factors such as schedules, track availability, and priority rules. Dispatching decisions must be made continuously and quickly, especially in high-traffic networks, making the problem both operationally critical and computationally challenging.

Despite significant technological advances in other areas of transportation and logistics, train dispatching remains heavily reliant on human decision-makers or static rule-based systems. This is largely because the problem is highly combinatorial: even with the smallest problems, there can be exponentially many possible sequences of decisions. Human dispatchers bring experience and intuition to these situations, but they are limited in how much information they can process and how consistently they can manage large-scale disruptions or optimize traffic flow over time.

Improving train dispatching systems has the potential to reduce passenger delays, minimize dispatching errors, and prevent deadlocks—where no train can proceed without violating physical, safety, or scheduling constraints. By optimizing dispatching we can also improve energy efficiency and capacity utilization, making rail transport more sustainable. In dense urban transit systems or busy freight corridors, even marginal improvements in dispatching can lead to significant gains in resource consumption or time savings.

Reinforcement Learning (RL) offers a new approach to tackling the TDP by framing it as a sequential decision-making problem. By formulating the TDP as an RL problem, an agent learns to make dispatching decisions through interactions with a simulated environment, similar to how people learn through trial and error. RL is an active area of research particularly well-suited for problems with delayed consequences, dynamic environments, and large state spaces—all of which apply to train dispatching.

This work focuses on the formulation of the Train Dispatching Problem for RL-based approaches, rather than on the algorithm implementation. This thesis emphasizes how the problem can be encoded as an RL task using graph structures to represent rail networks, define meaningful states, actions, and rewards, and manage constraints inherent to railway systems. The goal is to provide a robust and extensible framework that future researchers and practitioners can build upon when applying RL methods to train dispatching and related infrastructure scheduling problems. As demonstrated in recent research on applying reinforcement learning to car and pedestrian-level traffic, combing RL and graph structures can yield powerful results, so long as the algorithms and models are capable of modeling the underlying graphs [1].

To provide a thorough understanding of the approach, the remainder of this paper is organized as follows. In Section 2, we present the foundational background necessary for our formulation, beginning with a formal description of the Train Dispatch Problem, reinforcement learning, and graph neural networks. These concepts are then reinforced in a discussion of recent research that inspired this work. Section 3 details the proposed formulation, introducing Train Operation Graphs (3.1), Resource Conflict Graphs (3.2), and the state and action spaces (3.3, 3.4). Section 4 presents the outcomes of implementing the proposed reinforcement learning framework for the Train Dispatching Problem. It highlights the design of the graph-based DQN agent, the reward function employed, and the limitations observed during experimentation. Finally, Section 5 preliminary results are showcased from a graph-based Deep Q-Network (DQN) agent followed by future directions for work.

2 Problem Background

2.1 Train Dispatch Problem

The TDP is a well-known example of a large-scale combinatorial optimization problem, characterized by its complexity and the need for efficient decision-making under constraints. This problem becomes especially critical in dense networks with high traffic volume, where suboptimal dispatching can lead to cascading delays, congestion, or even deadlocks. Traditional approaches typically formulate TDP as a Mixed-Integer Programming (MIP) problem, where binary decision variables represent route allocations, resource occupations (e.g., track segments), and train movements over time. Constraints in this formulation encode physical infrastructure limitations, train precedence relations, safety buffers, and scheduling windows [2]. While MIP solvers can produce high-quality solutions, they are often limited by scalability, especially under tight time constraints or when dealing with real-time disruptions. Generally, the TDP presents a significant challenge for existing algorithms due to:

- Its high-dimensional decision space and tight operational constraints
- The need for fast and adaptive solutions under uncertainty

– The opportunity to drastically improve throughput, punctuality, and network resilience.

In practice, the TDP is currently managed either by human dispatchers or by static rule-based systems. Human expertise plays a crucial role in adapting to disruptions or prioritizing certain services, but manual dispatching lacks scalability and consistency. Rule-based systems, while automated, are quite inflexible and often suboptimal when conditions change. These limitations open the door for machine learning-based methods, particularly those capable of learning scalable policies from data and interaction.

The DISPLIB 2025 Train Dispatching Competition, which inspires this work, provides a standardized and open-source benchmark dataset for evaluating train dispatching algorithms [3]. DISPLIB scenarios model complex networks and real-world operational constraints, enabling rigorous benchmarking. Each instance in DISPLIB specifies a railway topology, time constraint information, and resource usage requirements. The formulation presented in Section 3 is deeply integrated with this standardized dataset.

Given the current lack of RL-based solutions and the standardized dataset provided by the DISPLIB competition, the TDP stands as a compelling application domain for Reinforcement Learning (RL) and Graph-based learning models.

2.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a subfield of machine learning that combines reinforcement learning (RL) with deep learning techniques to solve complex decision-making problems. RL focuses on training agents to make sequential decisions by interacting with an environment, receiving feedback in the form of rewards, and learning to maximize cumulative rewards over time. Deep learning, on the other hand, enables the representation of high-dimensional data through neural networks, making it possible to handle complex state and action spaces.

The seminal work by Mnih et al. [4] introduced the Deep Q-Network (DQN), which demonstrated the ability of DRL to learn policies directly from raw pixel inputs in Atari games. This approach used a convolutional neural network to approximate the Q-value function, enabling the agent to make decisions in high-dimensional state spaces. The success of DQN highlighted the potential of combining deep learning with RL to tackle problems that were previously intractable.

These advancements have established DRL as a versatile and powerful tool for solving sequential decision-making problems in diverse domains, including robotics, healthcare, and transportation systems.

Markov Decision Processes Markov Decision Processes (MDPs) form the backbone of reinforcement learning. MDPs offer a structured framework for modeling decision-making in environments where outcomes are influenced by both randomness and the agent's actions. An MDP is formally defined by the tuple (S, A, P, R, γ), where:

– S: A finite set of states representing all possible configurations of the environment.
– A: A finite set of actions available to the agent.
– $P(s'|s,a)$: The state transition probability function, which defines the probability of transitioning to state s' given the current state s and action a.
– $R(s,a)$: The reward function, which specifies the immediate reward received after taking action a in state s.
– $\gamma \in [0,1]$: The discount factor, which determines the importance of future rewards relative to immediate rewards.

In essence, an MDP is a process where the current state contains all the information necessary to determine the optimal action and its associated reward. In reinforcement learning, this process is typically modeled as an interaction between an agent and an environment, governed by underlying dynamics. At each timestep, the environment provides the current state s_t to the agent, which selects an action a_t. This action is then applied to the environment, which updates its state to s_{t+1} and computes the corresponding reward r_{t+1}. This iterative process is illustrated in Figure 1.

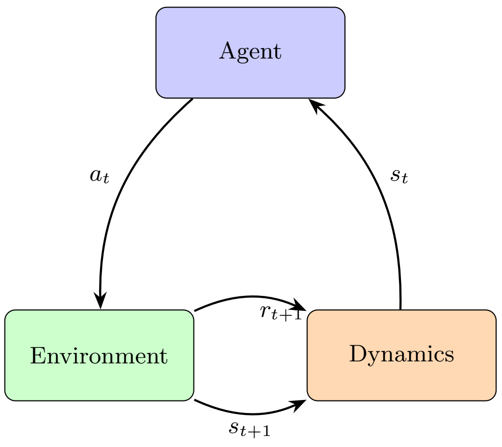


Fig. 1: The simplified markov decision process and reinforcement learning framework.

The goal in an MDP is to find a policy $\pi(a|s)$, which is a mapping from states to actions, that maximizes the expected cumulative reward, also known as the return:

$$G_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k})\right].$$

MDPs are widely used in reinforcement learning to model environments where agents learn optimal policies through interaction and feedback.

Deep Q-Network Deep Q-Networks (DQN) are a class of reinforcement learning algorithms that approximate the Q-value function using deep neural networks. The Q-value function, $Q(s, a)$, represents the expected cumulative reward of taking action a in state s and following the optimal policy thereafter. The DQN agent learns this function through experience replay and iterative updates.

The key components of a DQN agent are:

- **Q-Network:** A neural network that takes the current state s_t as input and outputs Q-values for all possible actions.
- **Target Network:** A separate network with weights θ^-, used to stabilize training by providing target Q-values. The target network is periodically updated to match the Q-network.
- **Experience Replay:** A buffer that stores past experiences (s_t, a_t, r_t, s_{t+1}). During training, mini-batches of experiences are sampled randomly from the buffer to break correlations and improve learning stability.
- **Bellman Equation:** The Q-network is trained to minimize the temporal difference (TD) error, defined as:

$$\delta = \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right],$$

  where γ is the discount factor.
- **Exploration vs. Exploitation:** The agent balances exploration (choosing new actions to explore the environment) and exploitation (choosing actions to exploit what it already knows) using an ϵ-greedy policy.

The training process involves iteratively updating the Q-network's weights θ using gradient descent to minimize the TD error. The learned Q-values guide the agent in selecting actions that maximize long-term rewards. For simpler tasks, a basic Q-network may suffice (such as a linear regression or heuristic model). However, for complex problems like the TDP, more advanced neural network architectures are likely better suited to handle the challenges effectively.

2.3 Graph Neural Networks

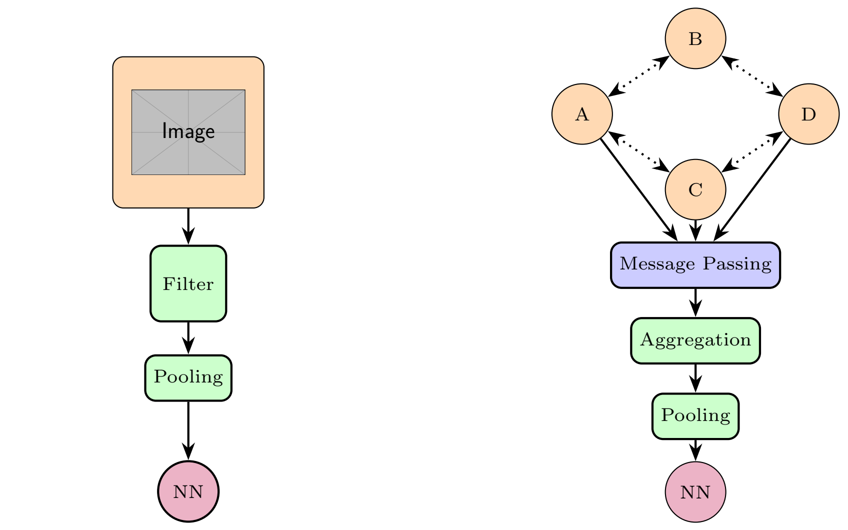
Graph Neural Networks (GNNs) are a class of neural networks designed to operate on graph-structured data, where entities are represented as nodes and their relationships as edges. GNNs leverage the graph topology and node features to learn meaningful representations for tasks such as node classification, link prediction, and graph classification.

GNNs extend traditional neural networks by incorporating message-passing mechanisms, where nodes aggregate information from their neighbors iteratively. This process enables the network to capture both local and global graph structures. The key components of GNNs include:

- **Message Passing:** Nodes exchange information with neighbors through a series of message-passing steps. At each step, a node updates its representation by aggregating messages from its neighbors.

- **Aggregation Functions:** Common aggregation functions include summation, mean, and max-pooling.
- **Update Functions:** After aggregation, node representations are updated using learnable functions, often implemented as neural networks.
- **Pooling Layers:** For graph-level tasks, pooling layers summarize the entire graph into a fixed-size representation.

Graph Neural Networks (GNNs) can be thought of as the graph-based counterpart to Convolutional Neural Networks (CNNs) for images. GNNs effectively aggregate information from neighboring nodes and edges, enabling each node to build a representation that incorporates both its own features and the context provided by its local graph structure. They then apply a pooling operation to extract and summarize key patterns and trends from the data. This comparison is illustrated in Figure 2



(a) Convolutional Neural Network (b) Graph Neural Network

Fig. 2: Comparison of CNN and GNN architectures.

Applications in Reinforcement Learning GNNs have shown significant promise in reinforcement learning (RL) for tasks involving graph-structured environments. Munikoti et al. [5] provide a comprehensive review of how GNNs enhance RL by enabling agents to reason over relational data, such as traffic networks or social graphs. They highlight the ability of GNNs to generalize across varying graph sizes and structures.

Graph Neural Networks (GNNs) have been successfully integrated with reinforcement learning (RL) to address complex graph-structured problems across various domains. For example, Almasan et al. [6] applied GNNs to network routing, demonstrating their ability to optimize routing decisions in dynamic environments. Tackling the problem of influence maximization in social networks, GraMeR [7] leverage GNNs combined with meta-reinforcement learning to learn transferable policies across diverse graph instances.

Devailly et al. [1] introduced IG-RL, an inductive graph reinforcement learning approach for large-scale traffic signal control. By utilizing GNNs, IG-RL efficiently manages massive traffic networks, enabling scalable and adaptive control strategies. Similarly, Agasucci et al. [8] introduced a DRL-based approach to the train dispatching problem, though their method is limited by the specificity of the dataset employed.

In summary, GNNs offer a robust framework for learning on graph-structured data, significantly augmenting the set of possible reinforcement learning applications. The ability of GNNs to learn from dynamic graph-structured data is a core concept behind this formulation.

3 Formulation

The following formulation introduces an approach to the TDP leveraging both graph-based representations and reinforcement learning techniques. By modeling train operations as Directed Acyclic Graphs (DAGs) and resource conflicts as Resource Conflict Graphs (RCGs), this formulation aims to capture the dependencies and constraints inherent to railway systems. The integration of GNNs is a crucial component that enables the reinforcement learning agent to process these graph structures effectively.

The remainder of this section is organized as follows. Subsection 3.1 introduces the concept of Train Operation Graphs in the DISPLIB format, detailing their structure and role in representing train operations and constraints. Subsection 3.2 describes the Resource Conflict Graphs, which model resource-sharing conflicts between trains. Subsection 3.3 defines the state space, highlighting how graph-based representations encode the environment's current state. Finally, Subsection 3.4 outlines the action space, specifying the set of actions available to the reinforcement learning agent for decision-making.

3.1 Train Operation Graphs

The Train Operation Directed Acyclic Graph (DAG) format provides a structured representation of train operations and their constraints. Conceptually, each train is modeled as a DAG, called the operations graph, where each node represents a possible operation the train can choose and each edge represents the transitions between operation. The primary components of the operations graphs are outlined as follows:

- **Nodes:** Represent operations, each with:
  - A **minimum duration** δ, indicating the shortest permissible time for the operation.

  - **Start time bounds** $[t_{\min}, t_{\max}]$, specifying the allowable time window for the operation to commence.
  - A set of **resources** required exclusively during the operation, each potentially associated with a **release time**. These resources typically represent the rail track section occupied by the operation.
- **Edges:** Define precedence constraints between operations, ensuring that dependent operations are executed in the correct order.
- **Entry and Exit Nodes:** The graph has a single **entry operation** (no incoming edges) and a single **exit operation** (no outgoing edges).

A visual example of a very simple train's operation graph can be seen in Figure 3. In this small example, the train has only one decision to make: choosing between operation 1 and operation 2. It then proceeds to operation 3, which has a minimum duration of 5, before moving to the exit operation.

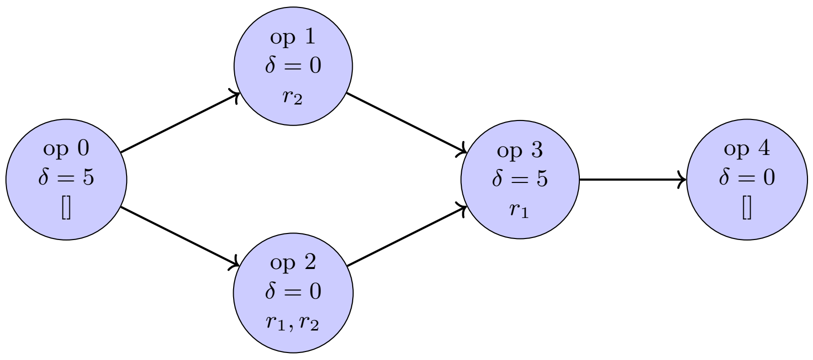


Fig. 3: Example Train Operation DAG with resource and duration constraints.

The Train Operation Graphs serve as the foundation for modeling the sequential decision-making process for one train. However, these graphs alone are insufficient to capture the interactions and conflicts between multiple trains sharing the same resources.

Resources and Constraints Resources represent entities (e.g., track sections) that cannot be shared between trains simultaneously. Operations requiring the same resource must satisfy the following:

- The **end event** of one operation must precede the **start event** of the next.
- The **release time** of the resource must be respected.

Feasibility Conditions A solution is feasible if:

1. Events are ordered chronologically.

2. Each train's operations form a valid path from the entry to the exit operation.
3. Start times respect the bounds $[t_{\text{min}}, t_{\text{max}}]$.
4. Operation durations meet or exceed the minimum δ.
5. Resource constraints are satisfied across all trains.

Objective Function The DISPLIB competition defines the objective as minimizing delay costs, calculated as:

$$v_i = c \cdot \max(0, t - t_{\text{threshold}}) + d \cdot H(t - t_{\text{threshold}})$$

where c and d are constants, t is the operation's start time, and H is the Heaviside step function [3].

Alternative objective functions can also be considered, such as minimizing the total time required for all trains to complete their respective exit operations.

3.2 Resource Conflict Graphs

To abstract resource labels from individual operations, this formulation introduces a new type of edge to represent resource conflicts between operations. These edges connect operations that share the same resource, transforming the disjointed train operation DAGs into a unified graph structure.

Figure 4 illustrates this concept, expanding on the toy example in Figure 3 with two trains. In this example, operation 2 of Train A utilizes resource r_1, which is also required by operation 3 of Train A and operation 1 of Train B. By introducing resource conflict edges, the formulation captures these dependencies and creates a "super graph" that integrates all train operation DAGs into a single, connected graph.

This unified graph serves as the foundation for the state space in the Markov Decision Process (MDP), encapsulating both individual train operations and their resource-sharing constraints. Such a representation enables the reinforcement learning agent to reason about both local and global interactions within the railway network.

3.3 State Space

In this formulation, the state space is represented as a Directed Acyclic Graph (DAG) composed of unconnected operation graphs. Each node in the graph corresponds to an operation and is characterized by a feature vector that captures relevant attributes of the operation. The node feature vector includes the following parameters:

- x_1: Train ID of the operation.
- x_2: Operation ID of the operation.
- x_3: Start lower bound of the operation (0 if not present).
- x_4: Start upper bound of the operation (10^6 if not present).

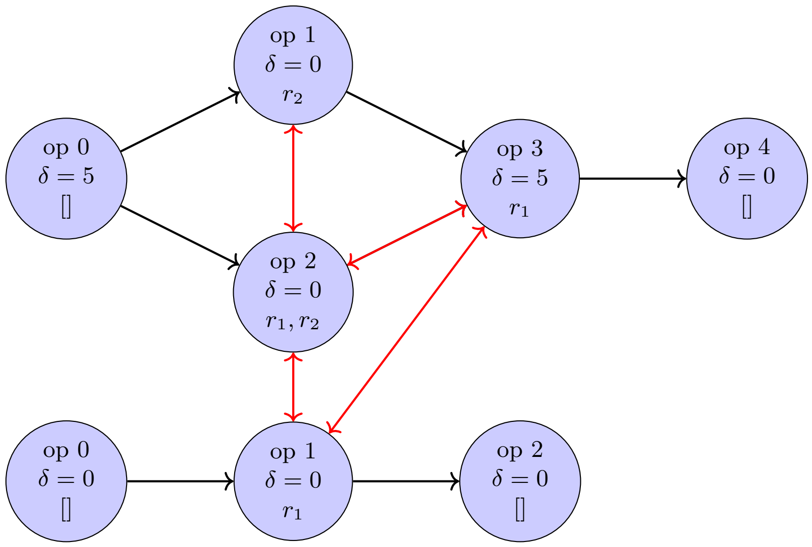


Fig. 4: Example Train Operation DAG with resource and duration constraints.

- x_5: Minimum duration of the operation (0 if not present).
- x_6: Can pick (1 if the operation has not been picked and can be picked, 0 otherwise).
- x_7: Currently picked (1 if a train has selected this operation and no successors exist, 0 otherwise).
- x_8: Picked (1 if the operation has already been picked, 0 otherwise).
- x_9: Not picked yet (1 if the operation has not been picked but can be picked in the future, 0 otherwise).
- x_{10}: Feasible (1 if the operation can be picked, 0 otherwise).
- x_{11}: Cannot be picked (1 if the operation has not been picked and cannot be picked in the future, 0 otherwise).
- x_{12}: Threshold of the operation (0 if not present).
- x_{13}: Haversine component of the operation (0 if not present).
- x_{14}: Linear delay of the operation (0 if not present).
- x_{15}: Operation start time (0 if not started yet).
- x_{16}: Operation end time (0 if not started yet).

Edges in the graph are assigned a feature vector that captures the type of relationship and resource constraints. The edge feature vector is defined as:

- e_1: Operation type (1 if this edge is between two train operations, 0 o/w)
- e_2: Resource type (1 if this edge represents a resource share, 0 o/w)
- e_3: Release time (the release time of the incoming operation's resource, 0 if there is no release time or if it is an operation type edge)

In addition to the node and edge features, the graph is also characterized by a global feature vector, denoted as $\mathbf{y}$. This graph-level feature vector contains a single entry, representing the current timestep of the solution. By including this global feature, the model gains a crucial awareness of the temporal context.

This representation provides a simple yet comprehensive structure for encoding the operations and their relationships, which can be processed by a Graph Neural Network (GNN) to generate a graph embedding for downstream decision-making tasks.

3.4 Action Space

The action space in this formulation consists of three primary actions that the agent can take at each decision step:

- **Add Node:** The agent selects a specific operation node from the graph to be added to the current schedule. This action is only valid if the selected node satisfies all feasibility conditions, such as respecting precedence constraints and resource availability. The agent must choose one node from the set of feasible nodes.
- **Discard Node:** The agent chooses to discard a specific operation node, indicating that the operation will not be scheduled. This action is irreversible and is typically used for operations that are deemed unnecessary or infeasible to include in the schedule.
- **Advance Timestep:** The agent advances the simulation a timestep, allowing the environment to update the state of operations and resources. This action is used when no immediate scheduling decisions can be made or when the agent determines that advancing time is the optimal strategy.

Each action is represented as a discrete choice, and the agent's policy maps the current state of the environment to one of these actions. The design of this action space ensures that the agent has the flexibility to make scheduling decisions dynamically while adhering to the constraints of the problem.

4 Results

4.1 Implemented Agent

The implemented solution features a graph-based DQN agent designed to operate on the nodes of the train operation graph. At each decision step, the DQN agent is provided with a node from the graph and outputs one of the three possible actions: *add node*, *discard node*, or *advance timestep*. If no node is deemed feasible at a given time, the environment automatically advances the simulation until the next decision point is reached. This ensures that the agent can continue making decisions without encountering unnecessary interruptions.

The full graph, which serves as the input to the agent, is processed by a GNN. The GNN computes embeddings for both nodes and edges, capturing the structural and relational information within the graph. These embeddings are then utilized by the DQN agent to output a solution, which attempts to effectively leverage the graph's topology and features to guide decision-making.

4.2 Reward Function and Limitations

The current reward function is designed to encourage the agent to compute feasible solutions efficiently. Specifically, the reward structure penalizes the agent for each decision it makes, incentivizing it to arrive at a solution in as few steps as possible. Additionally, the agent is rewarded for producing a feasible solution that adheres to the constraints of the Train Dispatching Problem.

However, a significant limitation of the current implementation is the absence of penalties for encountering deadlocks. Deadlocks, which occur when no further progress can be made without violating constraints, are critical to avoid in train dispatching scenarios. The lack of explicit penalties for deadlocks seems to lead the agent to overlook strategies that proactively prevent such situations. Future iterations of the reward function should address this limitation by incorporating penalties for deadlocks, ensuring that the agent learns to avoid gridlock.

5 Conclusion and Future Work

5.1 Future Work

This work opens several avenues for future research and development:

Improved Action Space Design The current action space, while functional, may not fully capture the complexities and nuances of the Train Dispatching Problem. Future work could explore more expressive and flexible action representations, such as hierarchical or multi-step actions, to better align with real-world dispatching scenarios. Additionally, incorporating domain-specific heuristics or constraints directly into the action space could improve the efficiency and feasibility of the learned policies.

Graph Reinforcement Learning Algorithm Development The integration of Graph Neural Networks (GNNs) with reinforcement learning (RL) has shown promise in handling graph-structured data. However, the development of specialized graph RL algorithms tailored to the unique characteristics of train dispatching remains an open challenge. Future research could focus on designing algorithms that leverage the structural properties of train operation and resource conflict graphs, enabling more efficient learning and better generalization across different network configurations.

Extensions to Other Applications While this work focuses on train dispatching, the proposed framework has the potential to be adapted to other domains involving complex scheduling and resource allocation problems. Examples include air traffic management, urban transit systems, and supply chain logistics. Extending the framework to these applications would require domain-specific modifications to the state and action spaces, as well as the underlying graph representations, but could significantly broaden the impact and utility of the approach.

Action Space for Constraint Reduction The current action space focuses on selecting operations or advancing the simulation timestep. However, an alternative approach is to redefine the action space to directly influence the constraint or problem space, enabling the agent to assist in solving the Train Dispatching Problem in conjunction with Mixed-Integer Programming (MIP) solvers.

In this modified action space, the agent's actions could be designed to:

- **Fix Routing Decisions:** The agent selects specific routing decisions for trains, effectively reducing the number of feasible solutions that the MIP solver must consider. For example, the agent could decide which track segment a train should use, thereby eliminating alternative routes from the problem space.
- **Prioritize Subproblems:** The agent determines which subproblems or regions of the railway network should be prioritized for optimization. This allows the MIP solver to focus computational resources on the most critical areas.

This hybrid approach has the potential to significantly improve scalability and solution quality, particularly for large-scale railway networks with complex constraints. Future work could explore the integration of this action space with state-of-the-art MIP solvers.

5.2 Conclusion

This honors thesis introduces a proof-of-concept reinforcement learning framework for the Train Dispatching Problem alongside preliminary results, emphasizing the formulation and potential of graph-based representations and deep learning techniques. By framing the problem as a Markov Decision Process and leveraging Graph Neural Networks, this work provides a foundational approach to train dispatching in the DISPLIB format that is both scalable and extensible.

As a formulation-focused study, this thesis aims to inspire future research in applying reinforcement learning to complex scheduling and resource allocation problems. Future efforts should prioritize refining the action space, developing tailored graph reinforcement learning algorithms, and exploring applications in other domains. These advancements will build upon the groundwork laid here, contributing to the broader understanding and development of intelligent transportation systems.

References

1. F.-X. Devailly, D. Larocque, and L. Charlin, "Ig-rl: Inductive graph reinforcement learning for massive-scale traffic signal control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 7496–7507, Jul. 2022. [Online]. Available: http://dx.doi.org/10.1109/TITS.2021.3070835
2. L. Lamorgese and C. Mannino, "The track formulation for the train dispatching problem," *Electronic Notes in Discrete Mathematics*, vol. 41, pp. 559–566, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571065313001418

3. Bjørnar Luteberget and Giorgio Sartor and Oddvar Kloster and Carlo Mannino, “DISPLIB: Train Dispatching Benchmark Library,” September 2024, dISPLIB 2025 Competition organized by SINTEF, announced at ODS 2024. [Online]. Available: https://displib.github.io/
4. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602
5. S. Munikoti, D. Agarwal, L. Das, M. Halappanavar, and B. Natarajan, “Challenges and opportunities in deep reinforcement learning with graph neural networks: A comprehensive review of algorithms and applications,” 2022. [Online]. Available: https://arxiv.org/abs/2206.07922
6. P. Almasan, J. Suárez-Varela, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio, “Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case,” *Computer Communications*, vol. 196, pp. 184–194, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366422003784
7. S. Munikoti, B. Natarajan, and M. Halappanavar, “Gramer: Graph meta reinforcement learning for multi-objective influence maximization,” 2022. [Online]. Available: https://arxiv.org/abs/2205.14834
8. V. Agasucci, G. Grani, and L. Lamorgese, “Solving the train dispatching problem via deep reinforcement learning,” *Journal of Rail Transport Planning and Management*, vol. 26, p. 100394, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210970623000264