POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Distributed Reinforcement Learning for Large-Scale Networks

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Giacomo Cartechini, 222464

Advisor:
Prof. Marcello Restelli

Co-advisors:
Gianvito Losapio
Marco Mussi
Alberto Maria Metelli

Academic year:
2023-2024

Abstract: Real-time, efficient management of network-based systems is a critical challenge in many domains. Approaches based on optimization have shown promising results, as they are able to provide optimal solutions by planning ahead, but this comes at the cost of recomputing the optimal solution every time an unpredictable event disrupts the planned schedule, which becomes unpractical when there are real-time constraints and the system has to scale to very large networks. In this work, we show how to build a formulation of a Markov Decision Process which is tightly related to the natural graph-based structure of problems of this class, with a special focus on the Train Dispatching Problem, but we will also briefly mention other domains where the same approach could be applied. Then we propose a distributed Multi-Agent Reinforcement Learning algorithm based on Q-Learning where agents are deployed to the nodes of the graph and, in a simulated environment of a railway network, we show how they are able to learn empirically optimal policies by acting independently and exchanging minimal information with their neighbors, even when unpredictable events such as train malfunctions occur.

Key-words: Multi-Agent Reinforcement Learning, Distributed Systems, Train Dispatching Problem

1. Introduction

Research in the field of Reinforcement Learning (RL) has seen a growing interest in the last years, both in academia and industry, due to the promising results that have been achieved in a wide variety of domains, such as robotics, games, finance, and many others. The natural ability of RL algorithms to learn from experience and adapt to changes makes them particularly suitable for control problems in stochastic environments, where planning ahead is difficult and the environment is affected by unpredictable events. In very large environments, where global solution methods fail due to computational complexity, distributed RL algorithms can fill the gap by allowing a group of independent agents to learn optimal control policies by acting locally and observing the effects of their actions on the environment.
In the last years, the demand for efficient management of large scale networks has increased significantly, making their optimal control a critical challenge in many domains. According to the Railway Statistics 2015 Report by the International Union of Railways (UIC) [2], European Union railways carry 416 billion *passenger-kilometers* (pkm) and 261 billion *tonne-kilometers* (tkm) of goods per year on a railway network of a total length of $350,000$ km, which make their management a complex task, of paramount importance for the economy of the continent. According to ENTSO-E, the European Network of Transmission System Operators, in 2022, 532 million customers were served by the European power grid, with a total of $312,693$ km of transmission lines, 3174.2 TWh of energy transmitted, and a total of $1,023,721$ MW of net generation capacity [1].

The operation of such large networks is a complex task where often decisions are still made by human experts, who have to take into account a large amount of information and make decisions in real time to avoid disruptions in the service. We are confident that, in the future, the use of AI techniques, and in particular Reinforcement Learning, will be able to provide assistance to human operators in the management of these networks, rendering their operation more efficient and reliable.
In this work, we use the expressions *network-based problems*, or *graph-based problems*, as umbrella terms to refer to a wide class of problems where the environment can be modelled as a directed graph. In particular we are interested in problems where the nodes of the graph can be seen as decision points, connected to other decision points by directed edges, along which the consequences of the decisions propagate. A wide variety of problems can be modelled in this way, such as the Train Dispatching Problem, a special case of the more general Vehicle Rescheduling Problem [11], which is the main focus of this work. But we will also mention, without diving into the details, other important problems on networks that are naturally modelled as graph-based problems, such as the problem of real-time management of power grids, or the problem of managing traffic flow in a city. Usually this class of problems presents a high degree of stochasticity, meaning that the environment is affected by random events that are not easily predictable, making planning ahead difficult. To face this challenge, solutions are required to adapt to rapid changes in the environment in real time, while also being able to scale to very large networks.
Classical optimization techniques have been successful, since they are able to provide optimal solutions by planning ahead, but they also present some limitations. In particular, to successfully build the optimization problem one needs to have a full model of the environment which is not always available, especially when dealing with very complex systems with high stochasticity. Moreover, the optimal solution is usually computed before the system starts to operate, and any unpredictable event that disrupts the planned schedule will require a complete replanning, which can quickly become unpractical to do real-time when dealing with very large instances of the problem.
On the other hand, Reinforcement Learning (RL) techniques, and in particular Multi-Agent Reinforcement Learning (MARL), are specifically designed to deal with real-time decision-making in stochastic environments, which makes them promising candidates in this scenario. Although less mature than classical optimization approaches, and being notoriously difficult to train, RL agents are able to observe changes or departure from the expected environment behavior, and they can learn to act optimally even in unforseen circumstances. Most RL techniques are model-free, meaning that they do not require a model of the environment to work properly, and they can adapt to changes in real time without needing to be retrained from scratch.

2. Related Work

Much work has been done in the field of optimization applied to graph-based problems. In the field of train dispatching, Lamorgese et al. [10] propose a decomposition of the problem which is solved by a master-slave approach, where a nearly optimal solution is found by iteratively solving the problem, adding new constraints at each iteration. Fischer et al. [8] propose a re-optimization approach where the solution of the problem is optimized again and again as disturbances occur in the system. Schälicke et al. [14] propose a formulation that can be solved in real-time by a MIP solver, and show that the method achieves acceptable computation times with good solution quality on a dispatching area in Germany.
In the field of power grid management, Jiang et al. [9] propose a risk based control strategy for power grid operation and solve the linearized problem with optimization techniques. Feng et al. [7] propose a MILP formulation of the power grid operation problem in order to achieve optimal allocation of electric power resources, adjusting power supply and demand.
Reinforcement Learning approaches applied to the train dispatching and scheduling problems include the one of Agasucci et al. [3], which propose both a centralized approach based on graph neural networks, and a decentralized approach under the assumption of full observability of the network. Yue et al. [19] propose a method to learn online dispatching policies which achieves online scheduling using idle time to solve future tasks.

3. Background

3.1. Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm which is applied to problems where an agent interacts with an environment over a sequence of time steps and has the objective of learning a policy that maximizes a certain scalar function, usually the cumulative discounted reward (return). In model-free RL

agents learn how to act optimally without needing a complete model of the environment, which is usually unavailable in real scenarios.

3.2. Markov Decision Processes

Markov Decision Processes (MDPs) [13] are a mathematical framework used to model decision-making problems where an agent interacts with an environment over a sequence of time steps. Formally, an MDP is defined by a tuple $\langle S, A, P, R, \gamma, \mu \rangle$ where:

- $S = \{s^1, s^2, \ldots, s^{|S|}\}$ is a finite set of *states*.
- $A = \{a^1, a^2, \ldots, a^{|A|}\}$ is a finite set of *actions*.
- P is the *transition probability function*:

$$\begin{aligned} P : S \times A \times S &\to [0, 1] \\ (s, a, s') &\mapsto P(s_{t+1} = s' | s_t = s, a_t = a) \end{aligned}$$

  P is the probability distribution of the next state of the environment s' given the current state s and the action a taken by the agent. In the case of MDPs, the Markov property holds, meaning that the transition probability of the next state depends only on the current state and action, and not on the past history of states and actions, more formally:

$$P(s_{t+1} = s' | s_t = s, a_t = a, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1} = s' | s_t = s, a_t = a)$$

- R is the *reward function*:

$$\begin{aligned} R : S \times A \times S &\to \mathbb{R} \\ (s, a, s') &\mapsto \mathbb{E}[r_t | s_t = s, a_t = a, s_{t+1} = s'] \end{aligned}$$

- $\gamma \in [0, 1]$ is the *discount factor*. The discount factor determines the importance of future rewards in the agent's decision-making process. A discount factor close to 0 makes the agent focus on immediate rewards, while a discount factor close to 1 makes the agent focus on long-term rewards.
- $\mu \in [0, 1]^S$ is the *initial state distribution*: $\mu_j = \mathbb{P}(s_0 = s^j)$

3.3. Return

The *return* at time t, G_t is the cumulative discounted reward an agent receives after time step t. Formally:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where r_{t+k+1} is the reward received by the agent at time step $t + k + 1$. In most RL problems, the return is the function that the agent aims to maximize.

3.4. Policy and Value Functions

Agents interacting with the environment use a *policy* to make their decisions. A policy π is a distribution over actions conditioned on the current state. Formally, a policy π is defined as:

$$\begin{aligned} \pi : S \times A &\to [0, 1] \\ (s, a) &\mapsto \pi(a|s) \end{aligned}$$

such that $\sum_{a \in A} \pi(a|s) = 1 \; \forall s \in S$.
An MDP is considered *solved* when the *optimal policy* π^* is found. The optimal policy of an MDP is the policy that maximizes the expected return. Formally:

$$\pi^* = \arg\max_{\pi} \; \mathbb{E}_{s \sim \mu} \left[G_t | s_t = s, \pi \right]$$

The *value function* of an MDP is defined as the expected return an agent receives when starting in a state s and then following policy π. Formally:

$$v^{\pi}(s) = \mathbb{E} \left[G_t | s_t = s, \pi \right]$$

The *state-action value function* of an MDP is defined as the expected return an agent receives when starting in a state s, taking an action a, and then following policy π. Formally:

$$q^{\pi}(s, a) = \mathbb{E} \left[r_t + \gamma G_{t+1} \mid s_t = s, a_t = a, \pi \right]$$

Given a state-action value function q, we can define the ϵ-greedy policy with respect to q as follows:

$$\pi_{\epsilon-\text{greedy}}(a|s) = \begin{cases} 1-\epsilon & \text{if } a = \arg\max_{a'} q(s,a') \\ \epsilon & \text{otherwise} \end{cases}$$

with $\epsilon \in [0,1]$.

3.5. Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) [4] is an extension of the standard RL paradigm to environments where multiple agents interact simultaneously. Each agent seeks to maximize its own reward function in a setting where agents must cooperate or compete with each other. In such environments, the actions of one agent can affect the rewards and state transitions of others, creating a need for agents to consider the strategies of their counterparts.
MARL introduces distinct challenges beyond those found in single-agent settings. In MARL, multiple agents interact within a shared environment, each aiming to optimize its own rewards. Key challenges include non-stationarity, partial observability, coordination and credit assignment.
A primary issue in MARL is *non-stationarity*. Since each agent learns and updates its policy over time, the environment becomes non-stationary from the perspective of any single agent. This makes it difficult for agents to learn stable policies, as the dynamics of the environment shift with the changing strategies of others.
Another significant challenge is *partial observability*, where agents have limited knowledge of the global state due to restricted observations. This can make it difficult for agents to make fully informed decisions, potentially leading to suboptimal actions. Additionally, communication protocols can be incorporated, allowing agents to share information and coordinate more effectively.
Another challenge is the *credit assignment problem*. In collaborative tasks where agents receive a global reward, it can be challenging to attribute success or failure to individual actions. This issue complicates each agent's ability to learn effectively from experience.
Managing the *exploration-exploitation trade-off* [6] is also more complex in MARL. Exploration by one agent may disrupt the learning of others, while excessive exploration across agents can destabilize the entire system.
The development of robust MARL systems requires addressing these challenges through methods that stabilize learning, support partial observability, scale efficiently, and foster effective cooperation among agents.

3.6. Temporal Difference Learning

Temporal Difference (TD) Learning [15] is an essential method in Reinforcement Learning (RL) for estimating value functions. TD learning updates value estimates incrementally at each time step. This incremental update allows TD learning to be applied in environments with long or continuous episodes, where waiting for an episode to complete is impractical.
In TD learning, value estimates are updated by *bootstrapping*, meaning that current estimates are refined using other, possibly incomplete, estimates rather than waiting for a final outcome. This approach enables agents to learn from incomplete sequences of experience and adjust value estimates in real time. The simplest form of TD learning, known as TD(0), updates the value of the current state $v(s)$ according to the rule:

$$v(s_t) \leftarrow v(s_t) + \alpha \left(r_{t+1} + \gamma v(s_{t+1}) - v(s_t)\right) \tag{1}$$

where:

- s_t is the current state,
- r_{t+1} is the reward received after transitioning from s_t to s_{t+1},
- $\alpha \in (0,1]$ is the *learning rate*, controlling the step size for updates,
- $\gamma \in [0,1]$ is the *discount factor*, determining the weight of future rewards.

The term $r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ is called the *TD error*, representing the difference between the expected and observed value estimates. By minimizing this TD error, the agent iteratively improves its value function estimates based on experience.
TD learning is particularly useful in situations where the agent operates in dynamic environments, as it allows for faster and more responsive updates than waiting until the end of an episode. TD learning is a foundational technique in RL, forming the basis for more sofisticated algorithms such as Q-Learning, SARSA, DQN, and others.

3.7. Q-Learning

The *Q-Learning* algorithm [18] is one of the simplest and most popular reinforcement learning algorithms. Q-Learning is used to learn the optimal state-action value function $q^*(s, a)$ of an MDP. Then, from the optimal state-action value function $q*$, the optimal policy π^* can be derived as follows:

$$\pi^*(s, a) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} q^*(s, a') \\ 0 & \text{otherwise} \end{cases}$$

The Q-Learning algorithm is illustrated here:

Algorithm 1 Q-Learning

Input: MDP $\langle S, A, P, R, \gamma \rangle$, learning rate $\alpha \in (0, 1)$, exploration $\epsilon \in [0, 1]$, number of episodes N
Preconditions: S finite, A finite
Initialize $q(s, a)$ $\forall s \in S, a \in A$ arbitrarily
for each episode **do**
  Observe state s, choose action a using ϵ-greedy policy induced by q, observe next state s' and reward r
  Update q:

$$q(s, a) \leftarrow (1 - \alpha)q(s, a) + \alpha \left[r + \gamma \max_{a'} q(s', a') \right]$$

end for
return q

Q-Learning is guaranteed to converge to the optimal state-action value function q^* under certain conditions [17]. We will use this algorithm as the basis for our tractation, and we will adapt it to become distributed across multiple agents.

3.8. Directed Graphs

A *directed graph* (or *digraph*) [16] is a mathematical structure used to model pairwise relationships between objects, where these relationships have a specific direction. Formally, a directed graph G is defined as an ordered pair $G = (V, E)$, where:

- V is a set of nodes, representing the entities in the graph.
- $E \subseteq V \times V$ is a set of directed edges, where each edge is an ordered pair (u, v) indicating a connection from node u to node v.

Unlike undirected graphs, the edges in a directed graph imply a one-way relationship. If $(u, v) \in E$, then there is a directed edge from u to v, but this does not necessarily imply an edge from v to u.
Some fundamental properties and concepts associated with directed graphs include:

- *Degree of a Node:* In directed graphs, each node has two associated degrees:
  - *In-degree* of a node v, denoted $\deg^-(v)$, is the number of edges directed into v.
  - *Out-degree* of a node v, denoted $\deg^+(v)$, is the number of edges directed out of v.
- *Paths and Cycles:* A *directed path* in G is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. A *cycle* is a directed path that starts and ends at the same node, with all other nodes in the path being distinct.
- *Strongly Connected Graphs:* A directed graph is *strongly connected* if there exists a directed path between every pair of nodes. For any two nodes $u, v \in V$, there is both a path from u to v and a path from v to u.
- *Acyclic Graphs:* A directed graph is *acyclic* if it contains no cycles. Such a graph is often referred to as a *Directed Acyclic Graph* (DAG).
- *Planar Graphs:* A graph is called *planar* if it can be drawn on a two-dimensional plane without any of its edges crossing, except at their endpoints (i.e., nodes). Formally, a graph $G = (V, E)$ is planar if there exists an embedding of G in the plane such that no two edges intersect except at their nodes.

3.9. Directed Multigraphs

A *directed multigraph* [5] is a generalization of a directed graph in which multiple directed edges between the same pair of nodes are allowed. In other words, a directed multigraph can have more than one directed edge from node u to node v, and these edges can be distinguished by labels.

Formally, a directed multigraph is defined as a pair $G = (V, E)$, where:

- V is the set of nodes, representing the entities in the graph.
- $E \subseteq V \times V \times L$ is the set of directed edges, where L is the set of edge labels. Each edge is an ordered triple (u, v, l), where $u, v \in V$ represent the tail and head nodes of the edge, and $l \in L$ is the label of the edge, used to distinguish different edges between the same pair of nodes.

In contrast to simple directed graphs, where an edge between any two nodes is either present or absent, a directed multigraph allows multiple edges with potentially different labels between pairs of nodes.

4. Problem Formulation

It is often very natural to formulate a transportation, communication or logistics problem as a problem of decision making in a graph. For instance, if we were to solve the problem of managing a railway network, we would consider the switches of the network as decision points, and therefore deploy agents to the railway switches to make decisions about train routing, with the goal of minimizing the total delay of the network.
On the other hand if we were to solve the problem of managing a power grid, we may consider power plants as decision points, and deploy agents to the power plants to make decisions about power generation, with the goal of minimizing the total cost of power generation, the total emissions, or the total power loss in the grid.
What all these problems have in common is that the network can be modelled as a graph where the nodes are decision points, and the edges are the connections between the decision points along which the effects of the decisions propagate.
If we were to model the problem of delivering packages in a city, we could consider the intersections of the city as decision points, and deploy agents to the intersections to make decisions about package routing, with the goal of maximizing the number of packages delivered in a given time frame. By observing what is happening in the neighborhood of the decision point (in the outgoing edges), nodes can make informed decisions about the best action to take. In the case of package delivery, an agent could choose to send a package to a neighbor node that is closer to the destination, or that has less traffic, or that has a higher probability of delivering the package on time.

4.1. The Markov Decision Process of a network

Let's represent our network as a directed graph $G = (N, E)$, where N is the set of decision points (nodes) of the network, and E is the set of connections between the decision points (edges). The choice of formulating the network as a directed graph derives from the fact that usually decisions "have a direction" (sending a package from a certain node to another, or routing a train from a certain switch to another). Anyway, note that this formulation does not cause any loss of generality: if we were to model a network where decisions are bidirectional, it would be sufficient to associate two edges to the same pair of nodes, one in each direction.

4.1.1 Local node observations

Each node $n \in N$ in the network has visibility of the state of the environment up to a certain depth d in the graph, which we will call the *observation depth*. Observation depth d poses a limit on the information that a node can obtain from the environment, and it is a parameter that has to be carefully tuned as it creates a trade-off between the amount of available information and the complexity of the learning process. As we will see later, it is important that nodes are able to observe the state of their successors in the graph ($d \geq 1$), since they will be affected by the immediate consequences of the decision taken by the node. For instance, if a node is controlling a switch in a railway network, it is important that it is able to observe the state of the tracks that are connected to the switch, in order to avoid collisions between trains, or to avoid sending a train to a track that is already crowded. A bad decision taken from the current node would increase the delay of the train, affecting the performance of the next nodes that will be responsible of routing that train in the future. An example of a local observation of depth $d = 1$ is shown in Figure 1.
We will call the set of all the local observations of depth d that a node n can make the *local state space* of the node, and we will denote it as $S_{n,d}$.

4.1.2 State space

The resulting state space of the global MDP representing the whole network is the combination of all possible local states of the environment observed by the nodes in the network. Given observation depth d, the generic

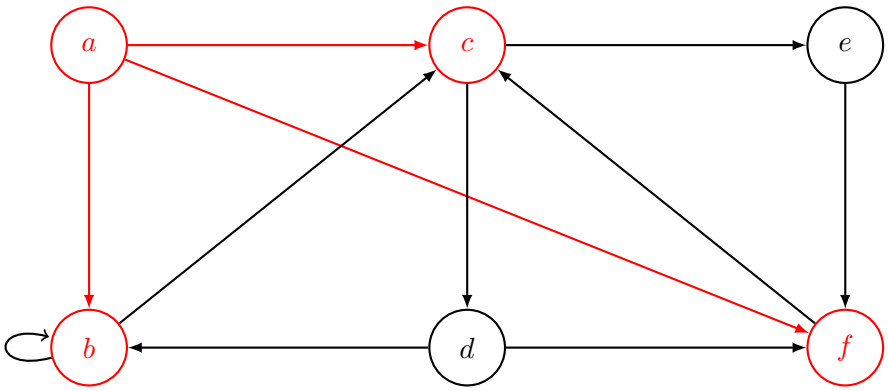

Figure 1: Edges and nodes included in observation of depth 1 from the point of view of node a, highlighted in red.

state s of the network can be represented by the following matrix:

$$s = \begin{bmatrix} \boldsymbol{s}_{d,1} \\ \boldsymbol{s}_{d,2} \\ \vdots \\ \boldsymbol{s}_{d,|N|} \end{bmatrix} \tag{2}$$

where each row $\boldsymbol{s}_{d,j}$ is the local observation of depth d made from node j. Formally, the state space S of the MDP representing the network is the Cartesian product of the local state spaces of all the nodes in the network, such that:

$$S = S_{1,d} \times S_{2,d} \times \cdots \times S_{|N|,d} \tag{3}$$

Note that if we formulate the MDP like this the state space becomes redundant, since the same information may be present multiple times basing on how deep the local observations are. As the observation depth d grows, nodes able to observe a larger portion of the network, increasing the overlapping between different local node observations. In the next sections we will show how redundancy does not cause any problem as we will depart from this formulation to generate a set of *reduced* MDPs, one for each agent in the network, that only contain a subset of the information present in the global MDP, with only the most relevant information needed for the agent to take a local decision.

4.1.3 Local actions and action space

As we mentioned before, each node $n \in N$ in the network is a decision point, that is, the node is required to make a local decision at each time step. Basing on the specific problem we are trying to solve, the set of actions that are available at each decision point may vary.
To maintain a general formulation, we will consider that each node $n \in N$ has a set of actions A_n available at each time step.
The action space A of the MDP representing the network is once again the combination of the action spaces of all the nodes in the network, such that:

$$a = \begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_{|N|} \end{bmatrix} \tag{4}$$

with each row $\boldsymbol{a}_j \in A_j$ being the action taken by node j, more formally:

$$A = A_1 \times A_2 \times \cdots \times A_{|N|} \tag{5}$$

4.1.4 Reward functions

The reward function R of such an MDP can be defined in many ways, and it ultimately depends on the specific problem we are trying to solve. It should be designed with the global objective of the network in mind, and it should be able to guide the agent towards the optimal solution.

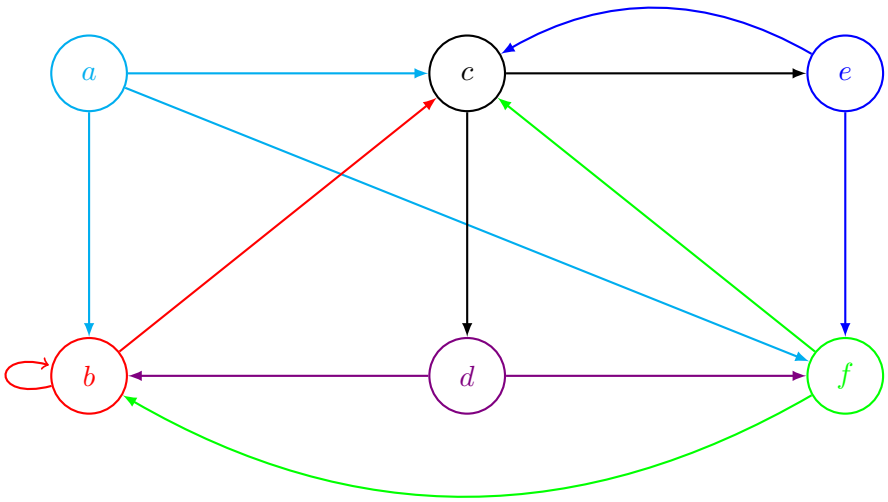

Figure 2: Partitioning of a network with 6 nodes into $k = 6$ partitions (each partition with its outgoing edges, shown in different colors).

In the case of the railway management problem, a possible reward function could be inversely proportional to increments in the total delay of the network, or it could be proportional to the number of trains that reach their destination on time. In the case of the package delivery problem, it could be proportional to the number of packages delivered at each time step.

4.2. Towards a multi-agent approach

One significant limitation of the global MDP we have just defined is that it is not scalable to large networks, since the state and action spaces grow exponentially with the number of decision points (nodes) in the network. Let's now show how we can decompose the global MDP into a set of *reduced* instances of that MDP, one for each agent in the network.

4.2.1 Partitioning the network

A *partition* of a set is a grouping of its elements into non-empty subsets, such that every element is included in exactly one subset.
We will partition the set of nodes N into k sets $N_1, N_2, \ldots, N_k$ such that:

$$N = N_1 \cup N_2 \cup \cdots \cup N_k \tag{6}$$
$$N_i \cap N_j = \emptyset \quad \forall i \neq j \tag{7}$$
$$N_i \neq \emptyset \quad \forall i \tag{8}$$

For our tractation, we will focus on the specific case of partitioning the network into $k = |N|$ partitions, where each partition consists of a single node.
In Figure 2 we show an example of partitioning of a network with 6 nodes.

4.2.2 The reduced MDP

After partitioning the network into $|N|$ partitions, one for each node, we can define a notion of *reduced MDP* with respect to each partition.
For a generic node $n_j \in N$, we define a new MDP with a smaller state space, which is equal to the space of local node observations of depth d, $S_{j,d}$, and a smaller action space, which is equal to the space of local actions, A_j. $S_{j,d}$ is derived from the global state space S by selecting only the information that is relevant to the specific node as follows:

$$\{n_j\} \rightarrow \underbrace{\begin{bmatrix} \boldsymbol{s}_{d,1} \\ \boldsymbol{s}_{d,2} \\ \vdots \\ \boldsymbol{s}_{d,|N|} \end{bmatrix}}_{\text{Global observation}} \rightarrow \underbrace{\boldsymbol{s}_{d,j}}_{\text{Local observation}}$$

Similarly, for the local action space A_j:

$$\{n_j\} \rightarrow \underbrace{\begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_{|N|} \end{bmatrix}}_{\text{Global action}} \rightarrow \underbrace{\boldsymbol{a_j}}_{\text{Local action}}$$

It is worth now to spend a few more words on the reward function of the reduced MDP. In the case of the global MDP it is often very easy and natural to formulate a reward function in terms of the global objective we are trying to optimize. Furthermore, as all the information of the environment is available at each time step, it is also easy to guide an agent towards the optimal solution.
In the case of a reduced MDP, the situation is more subtle. An agent operating in the reduced MDP has only a partial view of the environment, as it is not able to observe what is happening in the entire network, but only up to a certain depth away from the node making the observation; we can say that the environment is *partially observable* from the point of view of each agent. This means that each agent will never see the effects of its actions on the portion of the network that it cannot observe directly. This makes the formulation of the reward function crucial in guiding the agents towards the optimal solution, as it is the only source of information that potentially gives them a hint about the global effects of their actions. To make an example, if we were to solve the railway management problem, a possible reward function for the reduced MDP could be inversely proportional to the total delay of the trains that have been recently routed from the switch that corresponds to the node the agent is controlling. This way, even if the agent is not able to observe directly the state of the entire network, it is still able to understand the positive or negative effects of its actions on the global objective.
After partitioning the network and defining the reduced MDPs, we now have a set of $|N|$ reduced MDPs, one for each node, that can be solved independently by assigning an agent to each node. The agents will operate simultaneously and independently in their respective MDPs, and their actions will affect the global state of the network.
Since agents act in parallel, the complexity of the approach is not affected by the size of the network, since if we were to increase the number of nodes in the network, it would be sufficient to increase the number of agents accordingly, and the complexity of each reduced MDP would not change.

5. Learning on Graphs

Before describing the approach we propose to solve the MDP, we need to make some assumptions on how decisions taken on a specific node of the network affect other nodes, and how the effects of those decisions eventually propagate through the rest of the network.
It is safe to assume that in the class of problems we are considering, the effects of decisions taken on a node of the network will immediately affect the nodes that are directly connected to it, and eventually such effects will propagate through the network affecting the state of the nodes that are further away from the node that took the decision. Take for example the case of railway management, where a train is routed from a switch to one of the next switches. The decision taken by the agent which controls the first node will affect the future observation of the agent that controls the next node (the one receiving the routed train). The agent that controls the node that receives the train is not directly responsible for the decisions taken in the past, and should not receive any reward or penalty basing on the previous decisions of agents that control other nodes. On the other hand, the agent that took the decision to route the train should be held accountable for the consequences of its decision, and should receive a reward or penalty basing on the effects of its decision on the rest of the network, starting from the node that first receives the train. By doing so, agents are encouraged to take decisions that are beneficial for their neighbors, fostering cooperation among them.
In RL, the estimate of how good it is to be in a certain state is given by the value function $v(s)$, which is the expected return starting from state s, and the value function $v(s)$ can be estimated by the action-value function $q(s, a)$ by the following relationship:

$$v(s) = \max_a q(s, a) \tag{9}$$

If we had a way to estimate the action-value function q, we would therefore also have an estimate of the value function v, and we would also be able to calculate an ϵ-greedy policy π associated to q. For this reason in this work we focus on Q-Learning, a model-free RL algorithm that estimates the action-value function $q(s, a)$, and we try to adapt it to the specific needs that arise from our graph formulation.

5.1. Q-Learning on Graphs

Let us now consider a generic partition j, and denote with n_j the node associated to that partition. Agent j will be responsible for making decisions in the reduced MDP associated to partition j, given by the state space $S_{j,d}$ and the action space A_j.
At each time step t, agent j will observe the state $s_t \in S_{j,d}$ of the environment, take action $a_t \in A_j$, and observe the reward r_t. From our previous assumptions, the effect of action a_t will be immediately observed by the nodes that are directly connected to node n_j through its outgoing edges. We will denote with $F_j = \{n_1, n_2, \dots, n_{|F_j|}\}$ the set of nodes that are directly connected to node n_j through its outgoing edges. The effect of action a_t will be observed by the nodes in F_j, and they will communicate an estimate of their value function to agent j calculated from their q-table:

$$\hat{v}_i(s_{t+1}) := \max_{a'} q_i(s_{t+1}, a') \quad i = 1, 2, \dots, |F_j| \tag{10}$$

Agent j receives the estimates of the value functions of the nodes in F_j, and uses them to update its own q-table:

$$q_j(s_t, a_t) \leftarrow (1-\alpha) q_j(s_t, a_t) + \alpha \left(r_t + \gamma \sum_{i=1}^{|F_j|} w_i(a_t) \cdot \hat{v}_i(s_{t+1}) \right) \tag{11}$$

$$= (1-\alpha) q_j(s_t, a_t) + \alpha \left(r_t + \gamma \sum_{i=1}^{|F_j|} w_i(a_t) \cdot \max_{a'} q_i(s_{t+1}, a') \right) \tag{12}$$

The weights w_i are normalized and action-dependent, as they are used to give more or less importance to different value function estimates, depending on how much action a_t has affected each node in F_j. The values of the weights w_i are determined by the specific problem we are trying to solve, and they are not hyperparameters of the algorithm.
For instance, if we were to solve the train dispatching problem, we would associate each node to a switch in the railway network, and each edge as a track in the railway connecting two consecutive switches. In this specific case, when a node decides to send a train into a certain track, the effect of that decision will only affect the node that is directly connected to the specific edge associated to that track, therefore, the weight w_i associated to the value function estimate of the node receiving the train will be 1, and all the other weights will be 0, since the decision taken will not affect the other nodes connected to the switch. If we were to operate a power grid, we could associate each node to an active element of the grid, such as a power plant, and each edge as a power line. Let us assume that the only two available actions in this case are to increase or decrease the power generation of the power plant. If a node decides to increase its power generation, the effect of that decision will likely equally affect all the nodes that are directly connected to the power plant through the power lines, therefore the weights will be equal to $1/|F_j|$ for all the nodes in F_j.

5.1.1 Communication between agents

To implement this algorithm, nodes need to be able to communicate with their successors in the network, in order to exchange the value function estimates. The number of communication channels needed by each node to correctly update its q-table is $O(h)$ where h is the maximum out-degree of the graph representing the network. The total communication overhead of the network is equal, in the worst case, to the total number of edges in the network, which is $O(|E|)$.
Note that in the case of problems similar to train dispatching, as we mentioned before, the number of communication channels active at each time step is $O(|N|)$, since every node just needs to communicate with one other node, the only one that is directly affected by its decision.

6. A case study: railway networks and the Train Dispatching Problem

There are mainly two stakeholders involved in the operation of a railway network: the *Infrastructure Manager* (IM) and the *Train Operating Company(s)* (TOC). The IM is responsible for the maintenance and management of the infrastructure. It must schedule maintenance activities without significantly impacting the train services, which becomes increasingly challenging as the traffic on the network grows, since maintenance tasks will often conflict with train schedules. IMs have also the responsibility of deciding where to deploy the necessary resources (such as maintenance teams and equipment, vehicles, etc.) across a large network. To efficiently do this, they need to predict the future demand of the network to a certain extent.

On the other hand, TOCs are responsible for the operation of the trains. They rely on track availability to run their services, and plan their schedules according to the availability of the infrastructure provided by the IM. They are directly responsible for the passengers' experience and must ensure that trains run on time and that passengers are informed in case of delays or cancellations. TOCs need visibility on the network maintenance schedule to plan their services accordingly. When unexpected events occur, such as train malfunctions or delays, TOCs must be able to make real-time decisions on how to reschedule the trains (train dispatching) in order to minimize the effect of those events on the network.
We will now show how to apply our distributed MARL approach to the latter problem, that is the problem of real-time train dispatching in a railway network.

6.1. The Flatland Simulator

To test our approach, we are going to use the *Flatland simulator* (`https://flatland.aicrowd.com/intro.html`) which is a simple, yet powerful railway simulator designed to test reinforcement learning algorithms on the problem of real-time train dispatching on a grid-like environment.

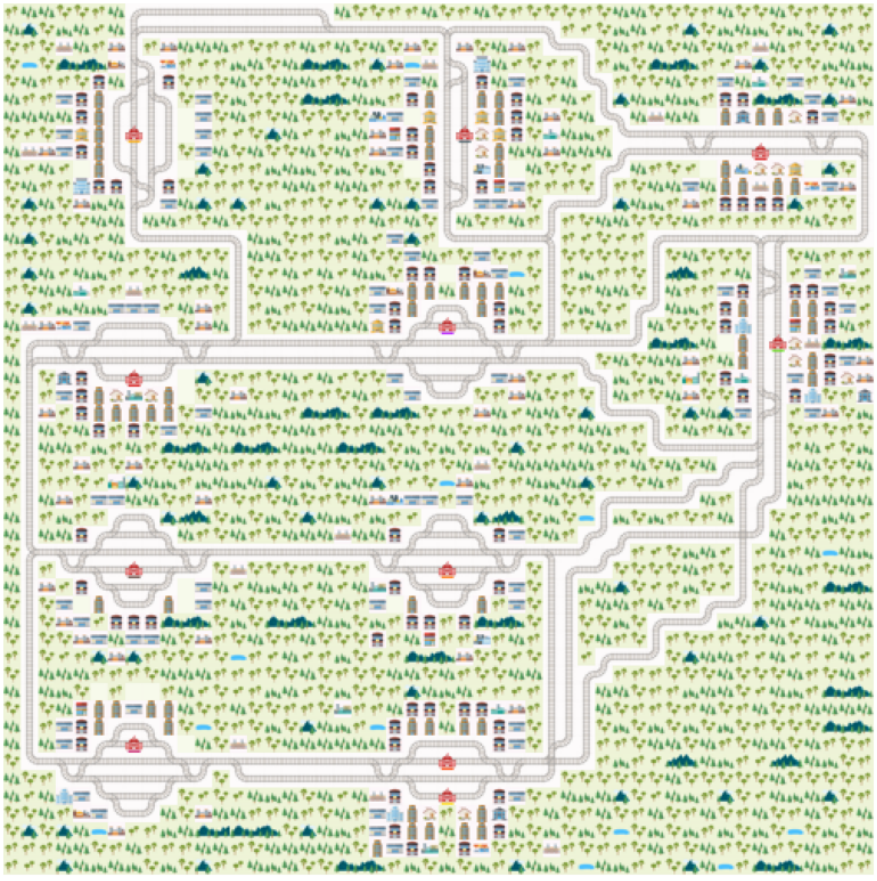

Figure 3: Flatland environment example

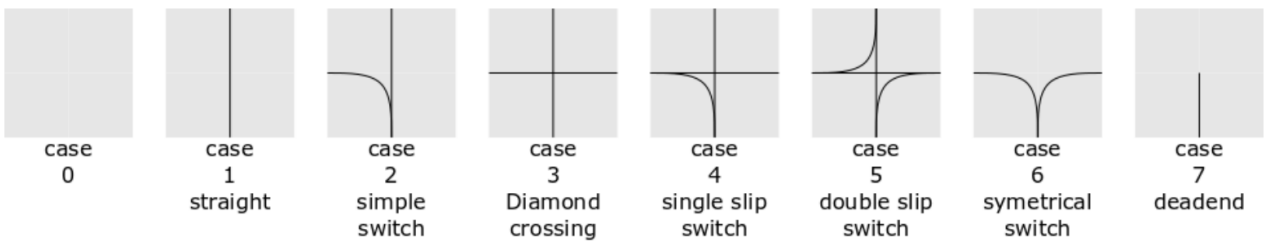

Figure 4: Flatland environment railway cell types

The core environment in Flatland is represented as a two-dimensional grid, such as the one in Figure 3, where

each cell can be one of the eight different types of cells shown in Figure 4, taking into account all the possible rotations and reflections of each cell type. Trains can move inside the grid only along the rails and they can make only the transitions allowed by the type of cell they are currently in. Take for example the cell type 2 of Figure 4, with the same orientation as shown in Figure 4, which allows trains oriented towards *North* to go *North* or *West*, trains oriented towards *East* to go *South*, and trains oriented towards *South* to go *South*. The *transition set* of that cell is therefore: $\{(N,N),(N,W),(E,S),(S,S)\}$. Transition sets will be later used to define the multigraph representation of the Flatland environment.
The grid of the Flatland environment can be randomly generated from a seed, and it can be configured with different parameters, such as the size of the grid, the number of trains, the number of cities in the map, the maximum number of parallel tracks inside cities, the maximum number of parallel tracks outside cities, and the random seed. Moreover, the environment can be configured to include stochastic events, such as train malfunctions, implemented as a probability of a train to break down at each time step. When a train breaks down, it stops moving for a random number of steps, and it is then repaired and resumes its journey. While a train is broken down, it blocks the track it is currently on, and other trains cannot pass through it. Any train is assigned a schedule when the environment is generated, which includes all the necessary information about the expected departure and arrival times of the train, the starting position and the target station. Trains cannot leave the starting position until their earliest departure time, and they should reach their target station before their latest arrival time, otherwise they will be considered late.
The goal of each train is to get to its target station on time, while avoiding collisions with other trains. The environment is considered solved when all trains reach their target stations on time.

6.2. Malfunctions

Although malfunctions are the only type of stochastic event admitted in the Flatland environment, they are useful to model a wide range of possible events that can occur in a real railway network, such as train breakdowns, signal failures, and track obstructions. The parameters that regulate the occurrence of malfunctions are fixed at environment generation time.
At each step every train has a probability of malfunctioning, such that the number of malfunctions of a certain train at each time step follows a Poisson distribution with parameter λ equal to the malfunction rate of the train. When a train malfunctions, it stops moving for a random number of steps given by a uniform distribution with minimum duration $d_{\min}$ and maximum duration $d_{\max}$ fixed at environment generation time. Formally, the number of malfunctions of a train at each time step, N_{malf} can be expressed as:

$$N_{\text{malf}} \sim \text{Poisson}(\lambda)$$

and the duration of each malfunction d is also a random variable expressed as:

$$d \sim \text{Uniform}(d_{\min}, d_{\max})$$

Under the reasonable assumption of independence between the malfunctions of different trains, we can model the total number of malfunctions $N_{\text{malf, tot}}$ at each time step in a network with n trains as:

$$N_{\text{malf, tot}} \sim \text{Poisson}(n\lambda)$$

6.3. Transition Maps

As already mentioned in Section 6.1, each cell in the Flatland environment allows a restricted set of transitions based on the orientation of the train with respect to the cell. Trains, at each time step, can be oriented towards one of the four cardinal directions: *North*, *East*, *South*, and *West*. Let's define an encoding for the cardinal directions:

$$\begin{aligned} &\textit{North} \to 0 \\ &\textit{East} \to 1 \\ &\textit{South} \to 2 \\ &\textit{West} \to 3 \end{aligned}$$

Given a cell of the environment with position (i,j), we call the *transition set* $\mathcal{T}_{ij}$ of that cell the set of all possible transitions allowed in that cell, such that:

$$\mathcal{T}_{ij} \subseteq \{0,1,2,3\} \times \{0,1,2,3\}$$

An alternative representation for transition sets is the *transition matrix* $\mathbf{T}^{ij}$, which is a 4×4 matrix where each row corresponds to the orientation of the train, and each column corresponds to the possible transitions from

that orientation. The matrix entries are always 0 or 1, basing on the existence of the corresponding transition. Formally:

$$\mathbf{T}_{kl}^{ij} = \begin{cases} 1 & \text{if } (k,l) \in \mathcal{T}_{ij} \\ 0 & \text{otherwise} \end{cases}$$

For example, a cell of type 2 in the Flatland environment (Figure 4), with the orientation shown in the figure, and position (i, j)=$(7, 5)$ has the following transition set:

$$\mathcal{T}_{7,5} = \{(0,0), (0,3), (1,2), (2,2)\}$$

and the corresponding transition matrix will be:

$$\mathbf{T}^{7,5} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

6.4. A train-centered model: the Flatland MDP

We call the way the standard Flatland MDP is formulated *train-centered* because agents are associated with trains, and observations and actions are defined with respect to each train. At each step, every train has 5 actions available, and the action space for each agent is:

$$A = \{\textit{MOVE FORWARD},\ \textit{MOVE LEFT},\ \textit{MOVE RIGHT},\ \textit{STOP MOVING},\ \textit{DO NOTHING}\}$$

A single agent formulation of the Flatland MDP would use as a state the entire grid (although Flatland does not impose any type of observation a priori, indeed it leaves the choice on how to build the observation up to the user), that is a global view of the enviroment, with all the information about the current positions of the trains, their directions, their earliest departure and latest arrival times, their current state, their speed and target stations. All this information can be concatenated to form a global observation vector, which will be the state of the environment.
The action space of the single-agent MDP would be the union of the action spaces of all the agents, therefore:

$$A_{\text{single agent}} = A^{\text{number of trains}}$$

At each time step, the agent observes the global state of the environment, generates the action vector, which is then converted to a dictionary of actions, one for each train, which get executed by the environment.
Flatland uses a sparse reward function, which means every train receives a reward different from zero only when it reaches its target station or at the end of the episode, when the time expires. If the train arrives at its target, the reward is $\min(0, \text{latest arrival} - \text{actual arrival})$. If the train does not reach its target, the reward is equal to the negative shortest path distance from the train position to its target station.
In the single agent case, after the step is executed in the environment, a reward for each train is emitted, and the sum of all the rewards is returned to the agent.
In this setting, it is easy to build the corresponding multi-agent MDP, since the only change that's needed is to make sure every agent controls a single train. Every agent now receives only the reward of the train it is controlling, and it seeks to maximize the discounted cumulative reward of that train.
The main difference between the two approaches is that in the single agent case the solution of the MDP is a single policy that controls all the trains, while in the multi-agent case the solution is a set of policies, one for each agent, that together control all the trains.
In the next sections we will see how to depart from the Flatland formulation of the MDP, and how to model this environment according to the MDP formulation we proposed in Section 4.

6.5. Departing from the Flatland MDP

One of the first things to notice when looking at the Flatland environment is that trains have often no choice but to go straight on (Figure 3). Railway networks are almost always sparse: cities have a lot of rails coming in and out, but the rails between cities are usually few. Decisions are made at switches, where trains can change direction and choose a different path. This is the reason why we decided to model the railway network, and consequently the environment, as a directed multigraph, as we will show in the next sections. In this formulation, agents are associated with nodes of the network, and they are prompted to make a decision whenever a train arrives at a node they are controlling.

6.5.1 Multigraph representation of the environment

The railway network of the Flatland environment can be modelled as a directed multigraph, where nodes correspond to railway switches and edges correspond to railway segments connecting two consecutive nodes. Nodes are the only points where trains can change direction, therefore they are the only points where agents can make decisions, while edges are decisionless segments where trains can only move forward. The choice of multigraphs instead of simple graphs is justified by the fact that there may exist multiple different tracks connecting two switches, that occupy different positions in the map, and they need to be treated differently even if they have the same source and destination nodes. This model of the environment is formulated in order to satisfy the following desirable properties:

1. *Nodes should have more than one outgoing edge.* Nodes with only one outgoing edge would have no choice to make on the available transitions as they are forced to send trains in the only available direction. It's desirable to associate a decision-making agent to each node, therefore every node must have at least two outgoing edges.
2. *Nodes should have no more than two outgoing edges.* This property is desirable because, combined with property 1, it implies a fixed size action space for each agent. While actions may have different meanings for different nodes, every node should have the same number of choices as this would greatly simplify both the formulation of the MDP and the implementation of the agents.

Keeping these properties in mind, let's define how the multigraph is generated.
Given a Flatland environment with grid width w and grid height h, define the following sets:

$$W := \{0, 1, \dots, w-1\}$$
$$H := \{0, 1, \dots, h-1\}$$
$$D := \{0, 1, 2, 3\}$$

In Flatland, given a train going through a switch at position (i, j) and orientation d, there are at most 2 available transitions, that means that $\sum_{d' \in D} \mathbf{T}^{ij}_{dd'} \leq 2$. The multigraph $G = (N, E)$ is defined as follows:

$$N := \{(i, j, d) \in W \times H \times D \mid \sum_{d' \in D} \mathbf{T}^{ij}_{dd'} = 2\}$$
$$E := \{((i, j, d), (i', j', d'), l) \in N \times N \times \{0, 1\} \mid g(i, j, d, l) = (i', j', d')\}$$

where g is a function that takes as input the node information (row, column, orientation) and a binary label, and returns the information about the closest neighbor following the first available direction if the label is 0, and the second available direction if the label is 1.
A multigraph built in this way satisfies both the properties formulated above.
The algorithm to implement multigraph generation is reported below.

Algorithm 2 Multigraph generation

$N \leftarrow \{(i, j, d) \in W \times H \times D \mid \sum_{d' \in D} \mathbf{T}^{ij}_{dd'} \geq 1\}$
$E \leftarrow \{((i, j, d), (i', j', d')) \in (N \times N) \mid (d, d') \in \mathcal{T}_{ij} \ \wedge \ d'$ leads to cell $(i', j')\}$
while $\exists \bar{n} \in N$ such that $|\{(n, n') \in E \mid n = \bar{n}\}| = 1$ **do**
  Pick any node $\bar{n} \in N$ such that $|\{(n, n') \in E \mid n = \bar{n}\}| = 1$
  Define the set $E_{\text{in}} := \{(n, n') \in E \mid n' = \bar{n}\}$ of incoming edges
  Define the successor node $\bar{n}'$ such that $(\bar{n}, \bar{n}') \in E$
  for each $(n_{\text{prev}}, \bar{n}) \in E_{\text{in}}$ **do**
    $E \leftarrow E \setminus \{(n_{\text{prev}}, \bar{n})\}$
    $E \leftarrow E \cup \{(n_{\text{prev}}, \bar{n}')\}$
  end for
  $E \leftarrow E \setminus \{(\bar{n}, \bar{n}')\}$
  $N \leftarrow N \setminus \{\bar{n}\}$
end while
return N, E

6.5.2 MDP Formulation

Now that we have defined the multigraph representation of the railway network, we can proceed with the formulation of the MDP. Given a railway network with m nodes, we define the local node observation. The local node observation is a vector $\begin{bmatrix} \xi & \tau & \beta_0 & \beta_1 \end{bmatrix}^T$ where:

- $\xi \in \{0, 1, 2, \ldots, k-1\}$ is the id of the target station of the train currently stepping on the node, where k is the number of target stations in the environment. A train is said to be stepping on a node if it shares the same row, column and orientation of the node.
- $\tau \in \{0, 1, 2, \ldots, \tau_{\max} - 1\}$ is the current delay of the train stepping on the node, discretized in $\tau_{\max}$ intervals.
- $\beta_0 \in F$ is a flag for the outgoing edge with label 0.
- $\beta_1 \in F$ is a flag for the outgoing edge with label 1.

The set of flags F may provide useful information about what is happening in an edge. In a simple formulation we may choose $F = \{0, 1\}$, where 0 means that the edge is free, and 1 means that the edge is occupied by a train. In a more complex formulation, F may contain more information, such as the number of trains on the edge, the speed of the trains, the direction of the trains, etc. There are just 3 types of action for each node:

- Action 0: send the train in the direction of the outgoing edge with label 0.
- Action 1: send the train in the direction of the outgoing edge with label 1.
- Action 2: keep the train in the node (stop action).

The reward function we crafted is different from the one in the Flatland environment, as it is designed to take into account multiple factors and to be less sparse, in order to make training easier.

- The *delay component* penalizes the agent for increasing the delay of a train.

$$\text{delay}(s, a, s') := \sum_{j=1}^{m} \tau(s) - \tau(s')$$

  We noticed that penalizing the agent for increasing the delay of a train speeds up the learning process.
- The *collision component* penalizes the agent for causing a collision with another train.

$$\text{collision}(s, a, s') := \begin{cases} -\text{collision penalty} & \text{if action caused a collision} \\ 0 & \text{otherwise} \end{cases}$$

- The *target component* rewards the agent for sending the train to its target station.

$$\text{target}(s, a, s') := \begin{cases} \text{target reward} & \text{if action caused a train to reach its target station} \\ 0 & \text{otherwise} \end{cases}$$

The reward function is defined as:

$$r(s, a, s') := \text{delay}(s, a, s') + \text{collision}(s, a, s') + \text{target}(s, a, s')$$

The discount factor γ is set to 1, as the environment is episodic and the maximum episode length is fixed. An episode terminates when all trains have arrived to their destination or when a collision occurs. There will be a more in-depth discussion about collisions in the next sections. When trains are not stepping on nodes, they move forward by default, as no decision is needed outside nodes, and trains cannot change direction in the middle of a segment.

7. Experiments and results

7.1. An environment without malfunctions

The following results are obtained by simulating the distributed Q-learning algorithm on an environment without malfunctions, with a 40×40 grid and 4 stations, as shown in Figure 5. We generate schedules with 5 trains. For this experiment, we use a 3-buckets discretization for the delay τ:

$$\tau = \begin{cases} 0 & \text{if train is on time} \\ 1 & \text{if delay is less than 20\% of the expected travel time of the train} \\ 2 & \text{if delay is more than 20\% of the expected travel time of the train} \end{cases}$$

We define the flags set $F = \{0, 1\}$, and:

$$\beta_j := \begin{cases} 0 & \text{if there are no trains coming in the node's direction in the edge with label j} \\ 1 & \text{if there is at least a train coming in the node's direction in the edge with label j} \end{cases}$$

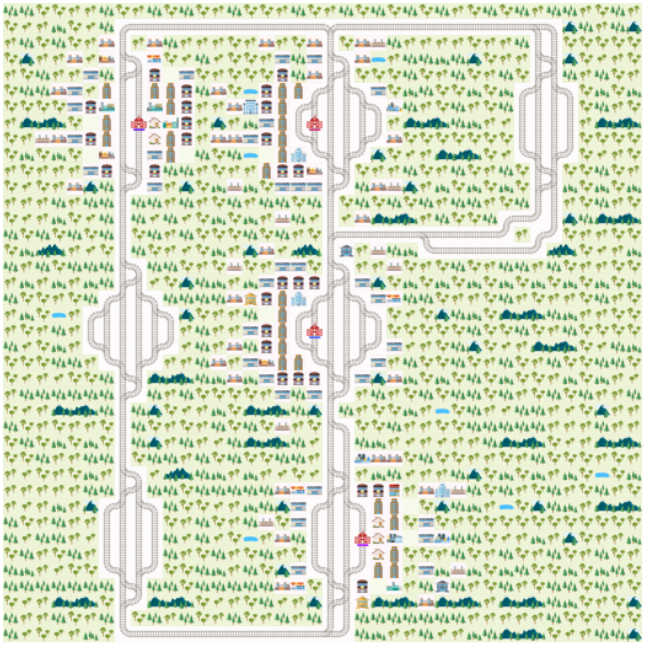

Figure 5: Flatland 40×40 environment.

The size of the state space for each agent in this specific case is:

$$|S| = \underbrace{4}_{\text{number of stations}} \times \underbrace{3}_{\text{number of delay buckets}} \times \underbrace{2}_{\text{number of flags}} \times \underbrace{2}_{\text{number of flags}} = 48$$

which is very small and manageable.
We also set:

$$\text{collision penalty} = 200$$
$$\text{target reward} = 200$$

The hyperparameters ϵ and α decay exponentially according to the formula:

$$\epsilon(t) := \epsilon_0 \cdot \epsilon_d^t$$
$$\alpha(t) := \alpha_0 \cdot \alpha_d^t$$

various decay rates are plotted in Figure 6.

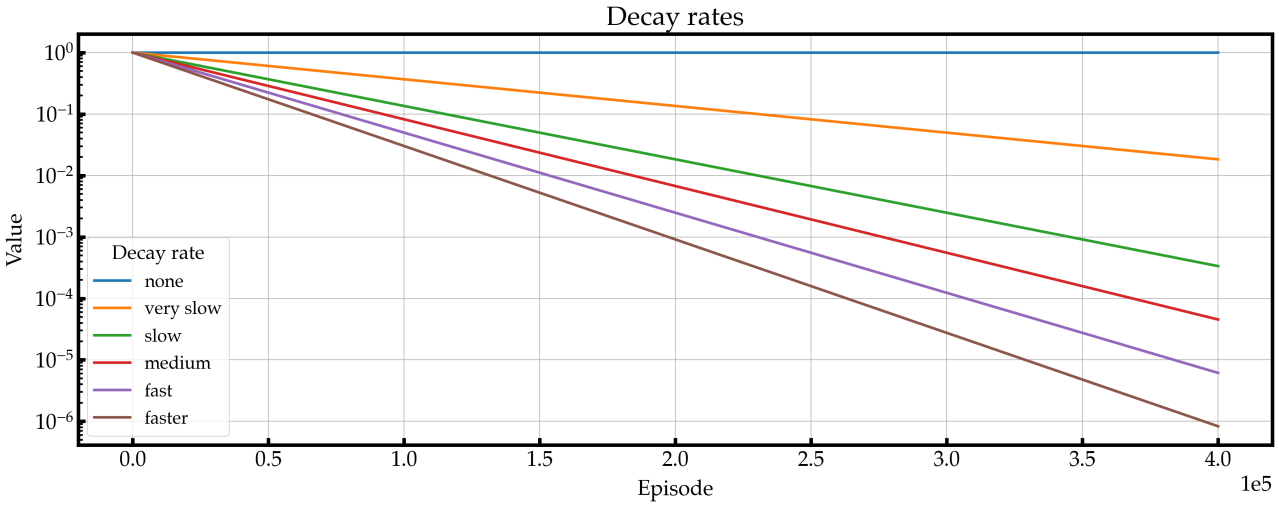

Figure 6: Various decay rates for α and ϵ.

We tried different hyperparameter configurations, and we show the training reward plots for the most promising ones in Figure 7.

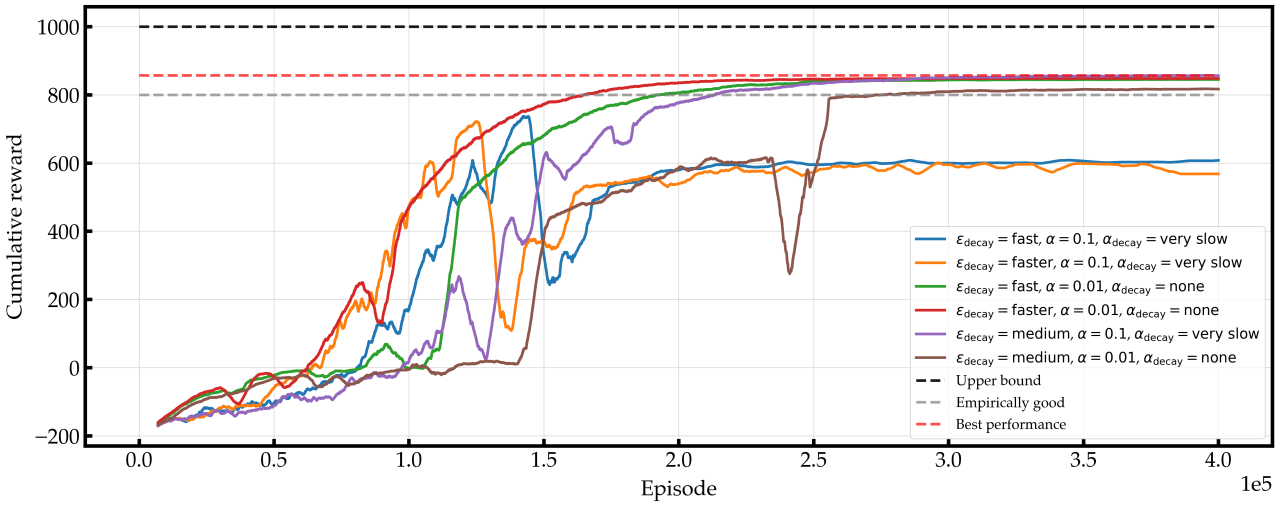

Figure 7: Training rewards for different hyperparameter configurations. Moving average over 7000 episodes.

Looks like the agents are able to learn a good policy in this environment, as there are hyperparameter configurations that are able to reach high rewards after seeing $\approx 2 \cdot 10^5$ episodes. In order to estimate how good this result is, we can measure the performance of the agents using the *trains on time* metric, which counts the number of trains arrived on time. In our environment with 5 trains, the upper bound of this metric is 5. As we can see in Figure 8, the best performing hyperparameter configurations in terms of rewards are also the best performing in terms of the *trains on time* metric.

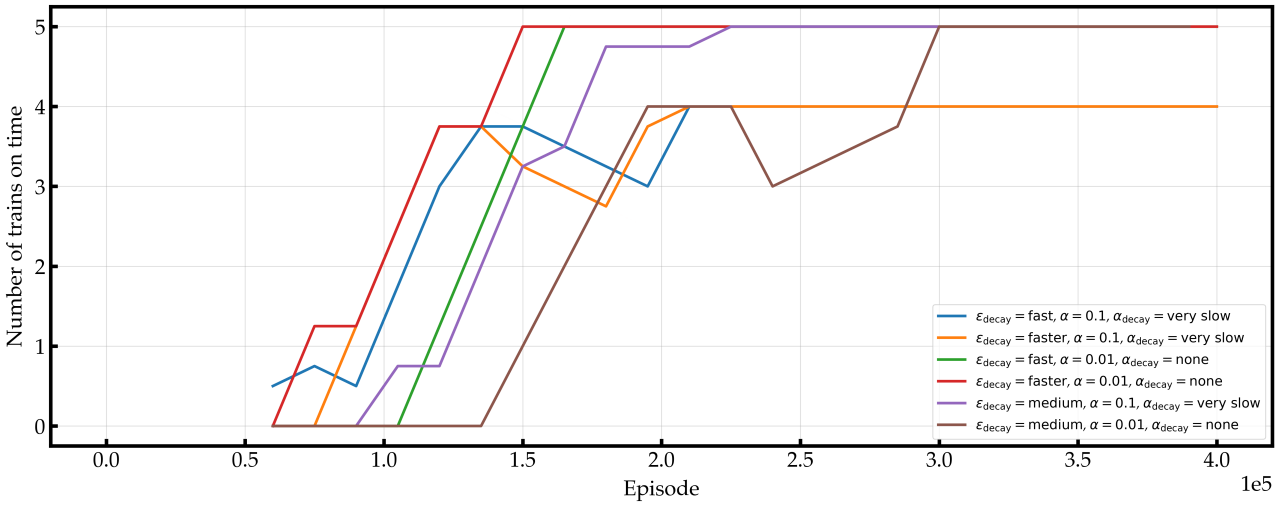

Figure 8: Number of trains arrived on time for different hyperparameter configurations. Moving average over 60000 episodes.

In the next sections we will gradually introduce malfunctions, and we will see how the agents are still able to learn a good policy unless the malfunction rates are extreme. In Figure 9, we show how the normalized errors of the q-values of each node with respect to the final q-values converge during training.

7.2. Dealing with deadlocks

Flatland's railway network environments are very sparse, just like the ones in the real world. By sparse we mean that there are clusters of nodes that are densely connected to each other inside cities, but these clusters are connected by a few edges to other clusters. In the real world, this is due to the fact that cities often have a central station where all the trains coming from the surrounding areas converge, and then they split again to reach their final destinations. It's not uncommon that two cities are connected by just a single railway line. What would happen if two trains are going towards each other on a single-track line? We call that situation a deadlock, because in Flatland trains stop at the point where they collide and they never move again until

the end of the episode, making the railway line unusable. This problem becomes impossible to solve in a pure multi-agent setting, as the agents have no idea of what the other agents are about to do, as they are not able to communicate their intentions to each other. Even if that kind of communication was possible, the agents would need some kind of protocol to decide which train has the right of way. This problem is very similar to the problem of simultaneous access to a shared resource in computer science, and a possible solution could be the same used in computer science: the use of locks. We could segment the railway network into critical sections, and associate a shared lock to each segment. A centralized entity would be in charge of managing the locks, and the trains would wait for the lock to be released before entering a critical section. In modern railway networks this is implemented by the use of signals, which are placed along the railway line and are controlled by a centralized entity, the railway traffic controller. This centralized entity would be transparent to the agents operating the nodes of the network, as the agent can act independently from the signals, but the signals will prevent trains from entering a critical section if there is another train already there. Therefore, from the point of view of the agents, locks would be just part of the environment. In this work our focus is on building a decentralized, multi-agent system that is able to learn how to operate a railway network, hence we will not consider the introduction of such a centralized entity, but if one wants to deploy a similar system in a real-world scenario, the introduction of a deadlock prevention mechanism is a must.
Anyway in the previous experiment we can notice how the agents are able to avoid deadlocks in the simple case of a malfunctions-free environment, as the flags β_0 and β_1 are informative enough for the agents to understand if there is the possibility of a deadlock associated with the corresponding edge.
On the other hand in the following sections we will deal with mulfunctions. Malfunctions can create unpredictable situations where the observation of the agents alone is not enough to prevent deadlocks, thus degrading the performance of the agents. Anyway we will show that in non-extreme cases the agents are still able to learn a good policy, even in the presence of malfunctions.

7.3. Introducing malfunctions

We will test our implementation on the same environment shown in Figure 5, but this time we will introduce malfunctions.
As it often happens in real railway network schedules, Flatland train schedule generation algorithm does not only take into account the time that is needed for a train to reach its destination following the shortest path, but it also takes into account the time that may be lost due to trains having to wait for other trains to clear the way. This is done to avoid generating unfeasible schedules, where it does not exist a solution that allows all trains to reach their destination on time.
Taking this into account, we created 4 different malfunction configurations, each one characterized by a different malfunction rate and duration. As explained in Subsection 6.2, there are 3 parameters that define a malfunction configuration:

1. The *malfunction rate*, λ, which is the probability that a train will have a malfunction at each time step in the Flatland environment.
2. The *malfunction minimum duration*, $d_{\min}$, which is the minimum number of time steps that a train will have to wait before the malfunction is resolved.
3. The *malfunction maximum duration*, $d_{\max}$, which is the maximum number of time steps that a train will have to wait before the malfunction is resolved.

We choose as malfunction rates $\lambda \in \{0.001, 0.005\}$, which means that on average a train will have a malfunction every 1000 or 200 time steps, respectively.
As durations we choose $(d_{\min}, d_{\max}) \in \{(5, 15), (15, 30)\}$. These durations are not chosen randomly, as 30 is exactly the margin of time that the Flatland enviroment gives to the trains to reach their destination, in addition to the shortest path time, for this specific environment configuration. These parameters allow us to create 4 different malfunction configurations, which span from the least to the most extreme case, as shown in table 1.

Configuration	λ	$d_{\min}$	$d_{\max}$
easy	0.001	5	15
normal	0.001	15	30
hard	0.005	5	15
extreme	0.005	15	30

Table 1: Malfunction configurations.

The results of our distributed Q-Learning algorithm, applied to the Flatland environment with the 4 different

malfunction configurations, are shown in Figure 10.
As we can see, in the easy and normal configurations agents are able to react to malfunctions and learn a good policy, minimizing the number of deadlocks as they learn and consistently being able to achieve the maximum number of trains arrived on time. In the hard configuration agents are still able to learn, but the number of deadlocks increases, as the agents are no more capable of preventing all of them. In the extreme configuration agents are not able to learn a good policy anymore, as the number of deadlocks is too high, mainly beacuse the high number of malfunctions contributes to clustering a high number of trains in a small portion of the network, making it nearly impossible for the agents to recover from the situation, and even in the few cases they are able to recover, they rarely manage to make trains arrive on time.

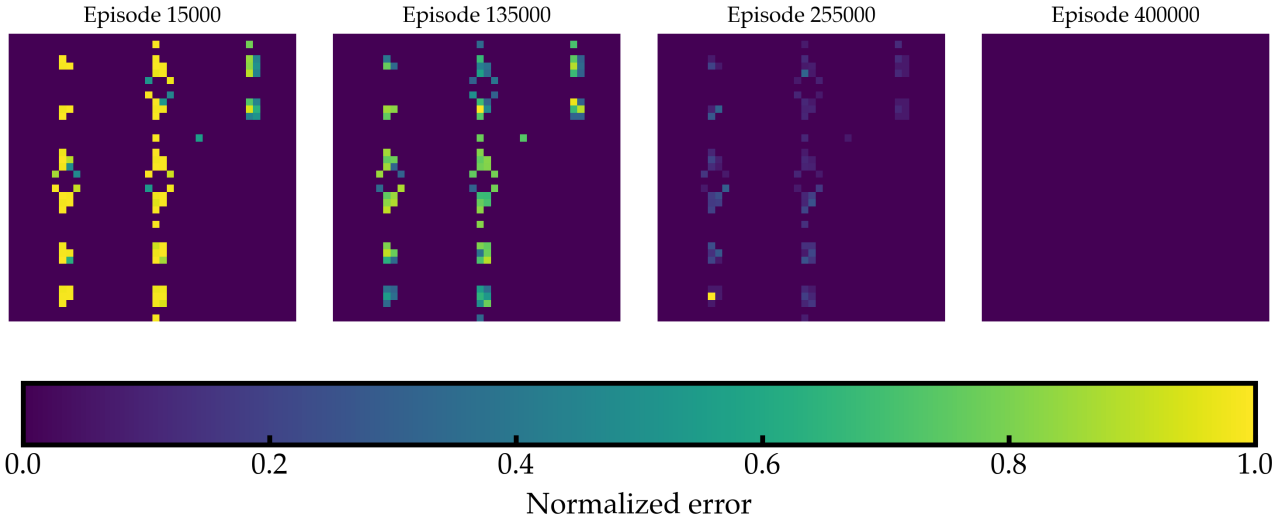

Figure 9: Node errors over the training process.

8. Conclusions

In this work we have shown how to model a Markov Decision Process around a graph-based problem, based on the assumption that nodes are decision points and edges are the connections between them, along which the effects of the decisions propagate. We proposed a scalable, distributed version of the Q-Learning algorithm, specifically adapted to this class of problems, which allows agents to learn independently with a minimal communication framework between neighbor nodes. Finally, we applied our algorithm to the Train Dispatching Problem using the Flatland simulator, and we showed how the agents are able to learn an empirically optimal policy, even in the presence of malfunctions, unless the malfunction configurations are too extreme. Our objective was to prove that a multi-agent approach would be viable even with an extremely simple observation and almost no external coordination among agents. It is very likely that a deeper local node observation would achieve even better results, despite the growth of the state space, as agents would be more aware of the local dynamics of the network, and they could become better at preventing deadlocks. Also we assumed that the partitioning of the network had to be done by assigning just one node to each partition. In the case of a generic partitioning of the network agents may control a subset of nodes, and not just one node. In this direction, some work has been done in the field of finding the optimal decomposition of an MDP on power grids (Losapio et al., 2024) [12], and it may be worth investigating if it is possible to apply these techniques to find the optimal partitioning of a railway network. Moreover, some techniques may be used to improve training speed and stability, such as the use of a replay buffer, which would allow agents to use past experience as well as new experience to make more stable updates to the Q-tables.

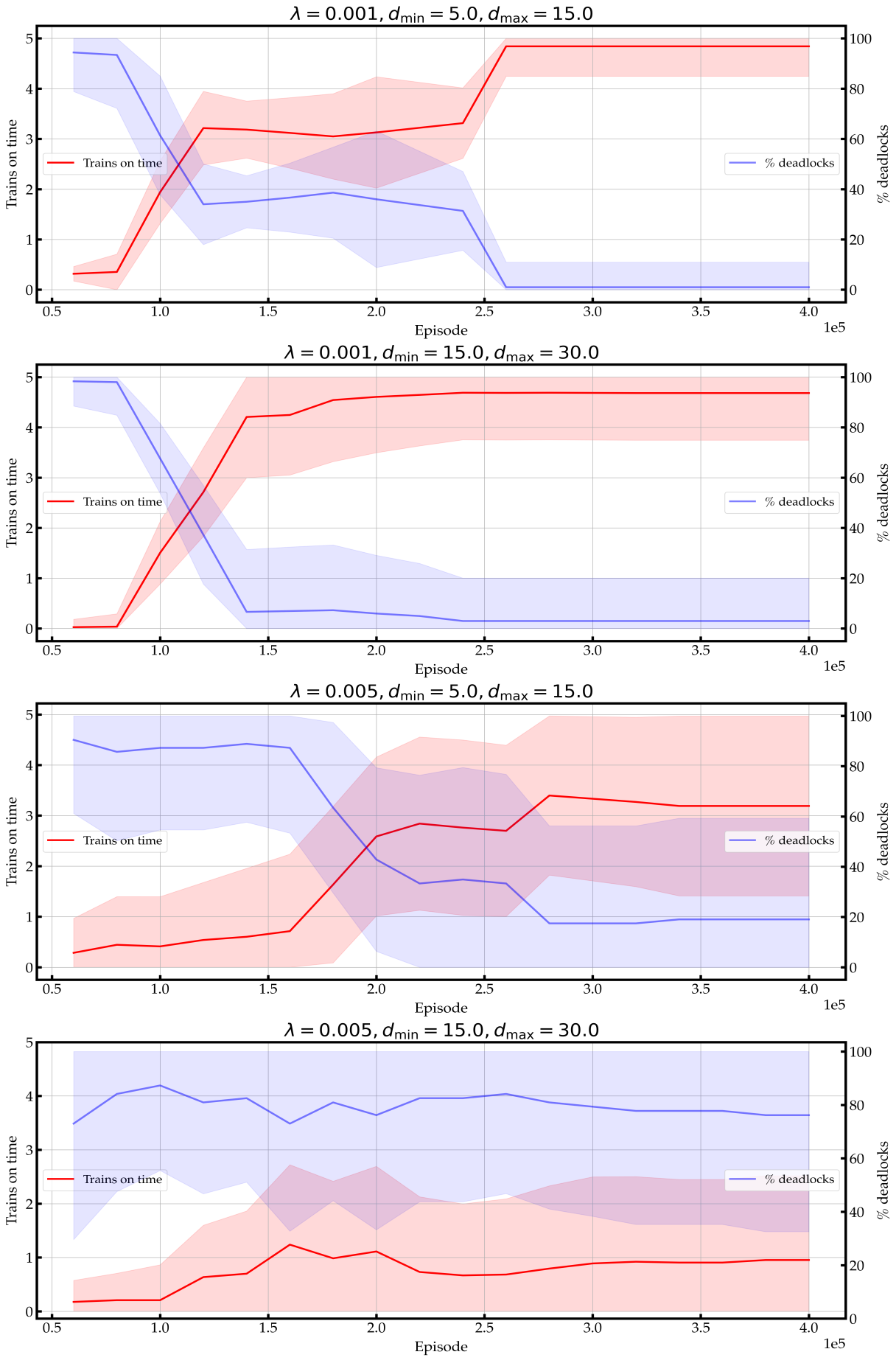

Figure 10: Training performance for the 4 different malfunction configurations.

References

[1] https://www.entsoe.eu/publications/.

[2] Railway statistics report. Technical report, International Union of Railways (UIC), 2015.

[3] Valerio Agasucci, Giorgio Grani, and Leonardo Lamorgese. Solving the train dispatching problem via deep reinforcement learning. *J. Rail Transp. Plan. Manag.*, 26(100394):100394, June 2023.

[4] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024.

[5] V K Balakrishnan. *Schaum's outline of graph theory: Including hundreds of solved problems*. McGraw-Hill Professional, New York, NY, February 1997.

[6] Oded Berger-Tal, Jonathan Nathan, Ehud Meron, and David Saltz. The exploration-exploitation dilemma: a multidisciplinary framework. *PLoS One*, 9(4):e95693, April 2014.

[7] Hua Feng, Chengxiang Zhang, Hanbing Zhang, Jichao Ye, Ning Ding, and Yonghai Xu. Research on the operation optimization of aggregate resources under the dual factors of power grid and economy. *E3S Web of Conferences*, 580, 10 2024.

[8] Frank Fischer, Boris Grimm, Torsten Klug, and Thomas Schlechte. A Re-optimization Approach for Train Dispatching. In Andreas Fink, Armin Fügenschuh, and Martin Josef Geiger, editors, *Operations Research Proceedings 2016*, Operations Research Proceedings, pages 645–651. Springer, April 2018.

[9] Tingting Jiang, Qianqian Xia, Yi Wang, Sheng Chen, and Zhinong Wei. Research on risk-based control strategy for power grid operation. In *2021 IEEE Sustainable Power and Energy Conference (iSPEC)*, pages 2815–2820, 2021.

[10] Leonardo Lamorgese and Carlo Mannino. The track formulation for the train dispatching problem. *Electron. Notes Discrete Math.*, 41:559–566, June 2013.

[11] Jing-Quan Li, Pitu B Mirchandani, and Denis Borenstein. The vehicle rescheduling problem: Model and algorithms. *Networks (N. Y.)*, 50(3):211–229, October 2007.

[12] Gianvito Losapio, Davide Beretta, Marco Mussi, Alberto Maria Metelli, and Marcello Restelli. State and action factorization in power grids. 2024.

[13] M L Puterman. *Markov decision processes*. Wiley Series in Probability & Mathematical Statistics: Applied Probability & Statistics. John Wiley & Sons, Nashville, TN, May 1994.

[14] Maik Schälicke and Karl Nachtigall. Solving the real-time train dispatching problem by column generation, 01 2024.

[15] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 8 1988.

[16] Richard J Trudeau. *Introduction to graph theory*. Dover Books on Mathematics. Dover Publications, Mineola, NY, 2 edition, February 1994.

[17] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[18] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[19] Peng Yue, Yaochu Jin, Xuewu Dai, Zhenhua Feng, and Dongliang Cui. Reinforcement learning for online dispatching policy in real-time train timetable rescheduling. *IEEE Transactions on Intelligent Transportation Systems*, 25(1):478–490, 2024.

Abstract in lingua italiana

L' efficiente gestione in tempo reale di sistemi basati su reti è una sfida critica in diversi settori. Approcci basati su ottimizzazione hanno mostrato risultati promettenti, in quanto riescono a trovare soluzioni ottimali, pianificando le azioni da intraprendere nel futuro. Tuttavia, questo tipo di approccio richiede il ricalcolo della soluzione ogniqualvolta eventi imprevedibili stravolgono la pianificazione originaria. Ciò può diventare impraticabile quando si ha a che fare con requisiti temporali stringenti e istanze molto grandi del problema. In questo lavoro si propone una formulazione di un *Markov Decision Process*, strettamente legata alla naturale struttura a grafo di problemi appartenenti a questa classe, con un'attenzione specifica al problema della gestione del traffico ferroviario in tempo reale; pur tuttavia, non mancheranno brevi riferimenti ad altri possibili campi di applicazione. Successivamente, si propone un approccio di *Reinforcement Learning* distribuito, basato su *Q-Learning*, dove gli agenti vengono associati ai nodi del grafo, e si mostra come, in un ambiente ferroviario simulato, siano capaci di apprendere delle *policy* empiricamente ottimali, agendo indipendentemente gli uni dagli altri e limitandosi a comunicare soltanto con i loro diretti successori nella rete, persino in presenza di eventi imprevedibili come malfunzionamenti.

Parole chiave: Multi-Agent Reinforcement Learning, Sistemi Distribuiti, Gestione di Reti Ferroviarie in Tempo Reale

Acknowledgements

Questa tesi rappresenta il culmine di un lungo e gratificante percorso di studio all'interno del Politecnico di Milano. Ci tengo a ringraziare i miei genitori, sui quali ho potuto sempre contare, per aver supportato le mie scelte e per avermi dato la possibilità di intraprendere questa avventura. A loro dedico la mia tesi. Ringrazio le mie nonne per il loro affetto, nonostante la distanza. Ringrazio Eugenia e tutti i miei amici, per essermi stati accanto in ogni momento. Un ringraziamento speciale va alla comunità politecnica tutta, che mi ha accolto nella città di Milano, e agli eccezionali professori che ho incontrato in questi anni.
Uno speciale augurio va ai miei amici che si laureeranno nei prossimi mesi, non vedo l'ora di festeggiare con voi.