POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Distributed Reinforcement Learning for Large-Scale Networks

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Giacomo Cartechini

Advisor: Prof. Marcello Restelli

Co-advisors: Gianvito Losapio, Alberto Maria Metelli, Marco Mussi

Academic year: 2023-2024

---

1. Introduction

As the demand for real-time, scalable and efficient techniques to manage network-based systems grows, graph-based problems are becoming increasingly interesting both to the research community and to the industry. In this work, by graph-based problems, or network-based problems, we refer to a wide class of problems where the environment can be represented by a graph, where nodes represent decision points and edges represent connections between decision points, along which the effects of the decisions propagate. Such problems include the Train Dispatching Problem in railway networks [2], where the goal is to react to real-time events that disrupt a predefined schedule, and act in order to minimize the overall delay of the network, or problems like the operation of a power grid in real time, or the dispatchment of a fleet of trucks with the goal of delivering goods in a urban environment. We will first propose a general model of such problems, where we will build a Markov Decision Process formulation which is tightly related to the underlying graph structure of the environment. We will then propose a distributed Reinforcement Learning algorithm based on Q-Learning [4] to solve it. The algorithm allows different agents to learn independently and asynchronously while exchanging a minimum amount of information between them. Finally we apply our algorithm to solve the Train Dispatching Problem in a simulated benchmark environment, and we show that the agents are able to learn an empirically good policy, even when there are events such as malfunctions that disrupt the normal operation of the network, unless such events are too extreme to be handled.

2. Problem Formulation

We start from a general model of a network-based system and we model it as a directed graph. We denote the graph as $G = (V, E)$, where V is the set of nodes and E is the set of edges. In this graph nodes correspond to the decision points of the environment we are operating in. For instance, if we were to operate a railway network, the nodes may correspond to network switches, and each possible decision may represent the specific track that a train should be routed to. If we were to operate a power grid, we could choose as decision points all the active elements of the grid, for example power stations, and we could model as decisions the amount of power that each station should produce. Edges represent the connections between decision points, and should be modelled in such

a way that the decisions taken at one node affect the state of the nodes connected to it. If there is no way the decision of a node can affect the state of another node, then there also should not exist an edge connecting them. Starting from this graph formulation, we want to deploy a Reinforcement Learning agent at each node, that will be responsible of controlling it in order to achieve a global objective in the network. Each agent can observe the state of the network up to a certain depth from its node, which we will call the *observation depth*, d, of the agent. A larger observation depth allows agents to make more informed decisions, but also increases the complexity of the problem. The state space of the MDP associated to each agent is then defined as the set of all possible observations of depth d from the node, and the action space is the set of all possible decisions that the agent can take at that node. At each time step each agent makes an observation, takes an action according to its policy, and receives a reward from the environment. The reward function is entirely problem dependent, but a few guidelines can be given to design it. As the agents are only able to observe part of the network, the reward is the only insight they have on the global objective, therefore it should be designed in such a way that if each agent is able to maximize its own reward function, then the global objective is somehow maximized as well. In the example of the Train Dispatching Problem, we would like to minimize the total delay of the network, therefore a possible reward function for each agent could be the increase of delay of the trains that have been recently routed from its node. Under the assumption that we use in this work, that is that decisions taken at a node immediately affect only nodes connected to them (and only later the consequences of their decisions propagate to the rest of the network), we can assume that even if the reward function is only based on local information, it can still be informative enough to learn a good policy. We will later show that this assumption holds in the experiments we conducted, as agents are able to learn an empirically optimal policy by acting locally and observing local effects of their decisions.

3. Q-Learning on graphs

We now propose a distributed version of the Q-Learning algorithm [4], specifically adapted for problems that can be formulated as we described in the previous section. The algorithm is based on the Q-Learning algorithm, which updates the q-value estimates of the agent at each time step, by following the rule:

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

where s is the current state observed by the agent, a is the action taken, r is the reward received, s' is the next state, α is the learning rate, and γ is the discount factor of the problem. Since we are in a multi-agent setting, where there is an agent controlling each different node of the network, we need to adapt the Q-Learning algorithm to allow agents to learn the effects of their decisions on their neighbors. Recall that it is possible to estimate the value function v of an agent given its Q-table as:

$$v(s) = \max_{a} q(s,a) \tag{1}$$

Let $n \in V$ be a generic node of the graph, and let $F_n = \{n_1, n_2, \ldots, n_{|F_n|}\}$ be the set of nodes such that $(n, n') \in E$, $\forall n' \in F_n$, that is the set of *successors* of node n in the network.

Decisions taken at node n affect the future state of all its successors, and we can use the value function estimate of the agents controlling nodes in F_n to estimate the update of the Q-values of the agent at node n.

Therefore, the update rule of the Q-values of agent controlling node n becomes:

$$\begin{aligned} q_n(s,a) \leftarrow & (1-\alpha)q_n(s,a) + \\ & \alpha\left(r + \gamma \sum_{i=1}^{|F_n|} w_{n_i}(a_t) \cdot v_{n_i}(s'_{n_i})\right) \end{aligned}$$

By applying equation 1 to the update rule, we obtain:

$$\begin{aligned} q_n(s,a) \leftarrow & (1-\alpha)q_n(s,a) + \\ & \alpha\left(r + \gamma \sum_{i=1}^{|F_n|} w_{n_i}(a_t) \cdot \max_{a'} q_{n_i}(s'_{n_i}, a')\right) \end{aligned}$$

That is, agent controlling node n will use a weighted average of the value function estimates

of the future states observed by the agents controlling nodes in F_n, instead of using its own estimate to update of its Q-table. The weights w_{n_i} are action-dependent and normalized, but most importantly they are not hyperparameters to be tuned, but they are determined by the problem at hand. Their use is to give more or less importance to successors of node n basing on how much actions taken at node n affect the future observation of each specific node $n_i \in F_n$. Take for example the Train Dispatching Problem, where nodes model switches in the railway network, and edges model the tracks connecting the switches. If a train is to be routed from the switch represented by node n to switch represented by node n', the the weight associated to n', $w_{n'}$, should be 1, and all the other weights should be 0. That is because the decision taken at n does not have direct consequences on other switches of the network, but only on the switch receiving the train. Conversely, in the case of power grid management, if we assume that nodes are power plants, and actions correspond to increasing or decreasing the power produced by the plant, then the weights associated to the nodes connected to the plant should be all equal, as the decision taken at the plant affects all the nodes connected to it in the same way, increasing the amount of power propagating from the plant to the neighbors.
This update formula has a very special meaning, since we are assuming that agents controlling successor nodes are able to provide a better estimate of the value function of the next state than the agent controlling the node itself. In problems belonging to the class we are considering, this assumption makes much sense, since the neighbors are the ones that are truly experiencing the consequences of previous decisions, and by propagating back the estimates of their value functions we are allowing information to flow backwards through the network, allowing agents to learn the long-term effects of their decisions even if they are not able to experience them by direct observation. By doing so, we give agents no responsibility about decisions made in the past, but they are fully accountable for the decisions they are taking now, since their actions will have direct consequences on the future states observed by their neighbors, and positive or negative effects will be reflected in the value function estimates of the neighbors, which are then propagated back to the agent itself in order to update its Q-values.

4. The Train Dispatching Problem

As a testbed for our algorithm, we solve the Train Dispatching Problem in the simulated environment provided by the Flatland simulator [1]. The Flatland simulator provides us with a grid-like environment, as the one shown in Figure 1.

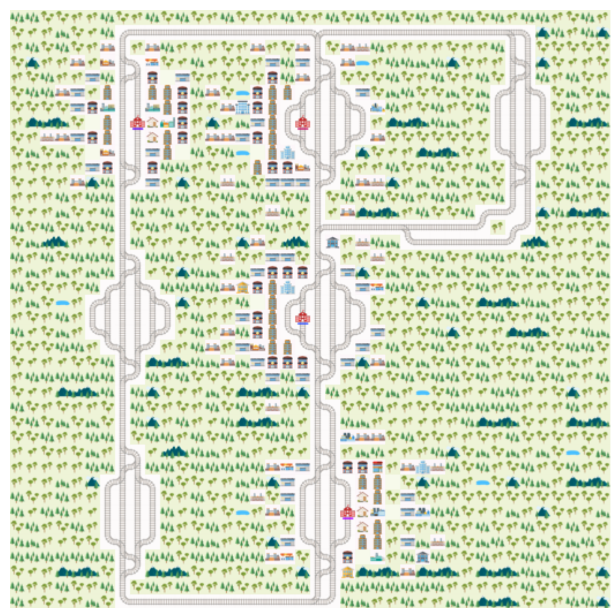

Figure 1: A snapshot of the Flatland environment.

At environment creation, the simulator generates an earliest departure and latest arrival time for each train. Our goal is to guide the trains from their starting position to their target station minimizing their delay.
First of all, we build a graph representation of the environment, where nodes represent railway switches and edges represent tracks connecting different switches. Our goal is to deploy a decision-making agent at each switch, that will be responsible of routing the trains stepping on the switch at its discretion in order to minimize the overall delay of the network. Each agent can observe the state of the network up to a certain depth d from the node, for our experiments we will set $d = 1$. We prefer to keep the observation space as small as possible in order to understand if agents can still learn given the limited amount

of information they can observe. The observation vector s of each agent will be:

$$s = \begin{bmatrix} \xi & \tau & \beta_0 & \beta_1 \end{bmatrix}$$

where ξ is the identifier of the target station of the train currently stepping on the switch, τ is the discretized delay of the train, and β_0 and β_1 are binary flags indicating if there is a train coming towards the switch in the corresponding outgoing track. In Flatland, given a specific train orientation, a switch can at most route a train to two different tracks due to the way the different topologies of switches implemented.
As we can see from Figure 2, for any type of switch, a train has at most two possible directions to go, therefore the action space of each agent is set to be of size 3: the agent can route the train to the left track, to the right track or stop the train (for some orientations trains have no choice other than moving forward, in that case no agent the corresponding node does not even exist in the graph). Trains that have no choice but to move forward, move forward by default.
The action space of each agent is therefore always of size 3: any agent can either route the train to the left track, to the right track, or stop the train. The reward function is designed in such a way that agents are rewarded negatively for increasing the delay of the trains or for causing collisions, and positively for sending trains to target stations. It is made of 3 components:

- The **delay component**, which penalizes agents for increasing the delay of the trains.

$$\text{delay}(s, s') = \tau(s) - \tau(s')$$

- The **target component**, which rewards agents for sending trains to their target stations. In the experiments we conducted, we set the this reward to be +200.

$$\text{target}(s, s') = \begin{cases} 200 & \text{if train sent to target} \\ 0 & \text{otherwise} \end{cases}$$

- The **collision component**, which penalizes agents for causing collisions between trains. In the experiments we conducted, we set this reward to be −200.

$$\text{collision}(s, s') = \begin{cases} -200 & \text{if collision} \\ 0 & \text{otherwise} \end{cases}$$

The total reward for each transition is then the sum of the three components:

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

5. Experiments

We first test our algorithm on an environment without malfunctions, with a 40×40 grid with 4 stations like the one shown in Figure 1. We generate schedules with 5 trains. The discount factor of the problem is $\gamma = 1$, since the environment is episodic and there is a maximum (high and fixed) number of steps after which the episode is terminated. The cumulative rewards of the agents during training are shown in Figure 3.
As we can see, agents are able to reach high rewards, but to really understand what is going on we need to use the *trains on time* metric. This metric counts the total number of trains arrived on time at their target station at each episode of training. The corresponding plots for this metric are shown in Figure 4.

5.1. Dealing with deadlocks

Flatland's railway networks are sparse, resembling real-world railways, where dense city clusters are connected by a few edges. Deadlocks occur when two trains collide on single-track lines, rendering the track unusable. In centralized systems, such issues are mitigated using signals managed by a traffic controller. However, this work focuses on decentralized multi-agent systems, and we will not implement such mechanisms. Right now, agents avoid deadlocks in malfunction-free scenarios using informative flags β_0 and β_1. In the next experiments we introduce malfunctions. As we will see, malfunctions create unpredictable situations, where observations alone may not suffice to prevent deadlocks, reducing agent performance. Despite these challenges, agents can still learn effective policies in non-extreme malfunction scenarios, demonstrating robust adaptability.

5.2. Introducing malfunctions

In Flatland, malfunctions are random events that can be configured by 3 parameters:

- The **malfunction rate**, λ, that is the average number of malfunctions of a train at each time step. The number of malfunction

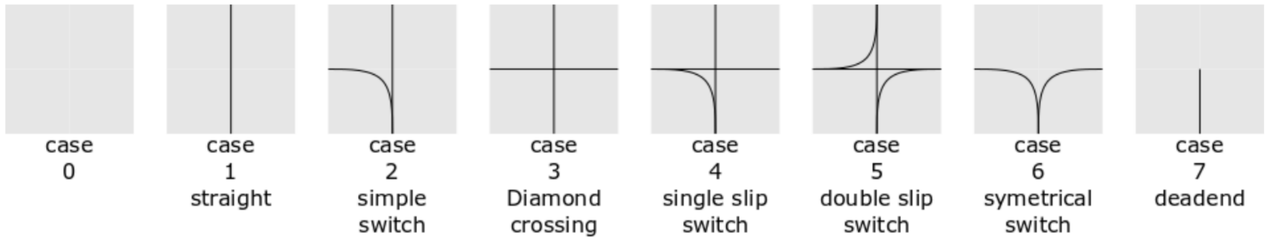

Figure 2: The different types of switches implemented in Flatland.

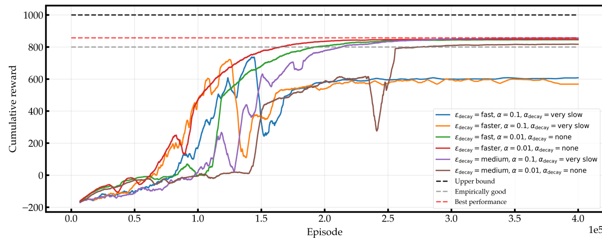

Figure 3: Cumulative rewards of the agents during training in an environment without malfunctions. Different training parameters are shown, the starting ϵ is always 1.0. Moving average over 5000 episodes.

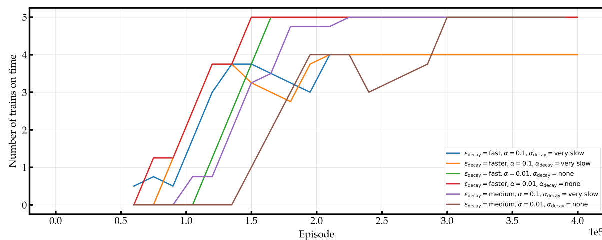

Figure 4: Number of trains arrived at their target station on time at each episode of training in an environment without malfunctions. Same hyperparameters as in Figure 3. Moving average over 60000 episodes.

of a train is defined by a Poisson distribution with parameter λ, Pois(λ).

- The **minimum duration**, $d_{\min}$, and **maximum duration**, $d_{\max}$, of a malfunction event, that are the minimum and maximum number of time steps a train will be blocked by a malfunction. The duration of a malfunction is defined by a uniform distribution between the two parameters, $\mathcal{U}(d_{\min}, d_{\max})$.

We will test our algorithm on the same environment as the previous experiment, but with different malfunction configurations, from the lightest to the most severe one, which are 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 are shown on the plots of Figure 5. 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.

6. Conclusions

In this work, we 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

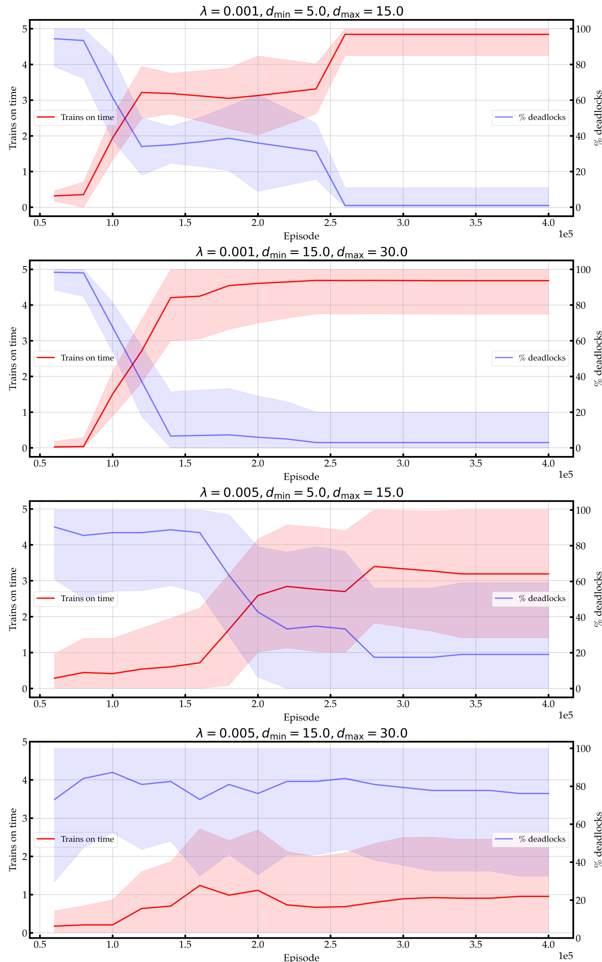

Figure 5: Training performance for the 4 different malfunction configurations of Table 1.

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. We also 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 [3], 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.

References

[1] https://flatland.aicrowd.com/intro.html.

[2] 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.

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

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