Reinforcment Learning Project

Elie Bosle , Marwan Kaddour , Raphaël Le Blanc

December 21, 2024

Abstract

We chose the application track for the project. Our application is the planning and affectation of platform in a train station .

Project link

https://drive.google.com/drive/folders/1rObmxeSSNtX2X7hSFxBprBc_NiVbBZNS?usp=share_link

Problem Modeling

Station Configuration

The station is defined by a set of **mainline tracks** L and **platform tracks** Q.

Routes

- Arrival itineraries I_a: Connect a mainline track to a platform track.
- Departure itineraries I_d: Connect a platform track to a mainline track.
- $I = I_a \cup I_d$.

An itinerary is identified by the tuple (id, departure, mainline track, platform track), as shown:

id	departure	mainline	platform
i	True	ℓ_i	q_i

Table 1: Example of an itinerary that corresponds to a departure from platform q to the mainline track l

Trains

A set T of trains. Each has the following parameters which can be found in the Table 2:

- An id t
- A direction (departure as True, arrival as False)
- A predefined mainline track ℓ_t,
- A variable platform track q_t,
- A type of circulation such as TGV, TER, etc., noted c_t,
- A list of its materials types m_t.

id	departure	mainline	platform	circulation type	materials types
t	False	ℓ_t	q_t	c_t	m_t

Table 2: Example of an arriving train assigned to platform q_t from the mainline ℓ_t

Constraints and Elements

1. **Platform Compatibility:** Assigning an itinerary i to a train t is placing the train t to the platform track q_i, ie $q_t \leftarrow q_i$.
   When assigning, the itinerary has to be compatible with the train's direction and its mainline (Compatibilities between Table 1 and Table 2). The constrains to verify are

$$\begin{cases} \ell_t = \ell_i \\ d_t = d_i \end{cases} \quad \text{with } d, \text{ the departure and } \ell \text{ the mainline track}$$

2. **Authorized platforms:** Some platforms q are inaccessible from certain mainline tracks L_q, materials types M_q or circulation type E_q. The constrains can be written as

$$\begin{cases} \ell_t \in L_q \\ \text{or, } m_t \in M_q \\ \text{or, } c_t \in E_q \end{cases} \iff q_t \neq q$$

3. **Route Incompatibilities:** Some trains arrive or depart around the same time and can introduce some delayed and so penalties. We get a penalty c_j for a train t_j and an itinerary i_j if $q_{t_j} = q_{i_j}$. In practice, the constrains given by the subject are in thousands, so we divide them by 1000 in order to get smoother rewards.

4. **Unassigned Trains:** Trains can be left unassigned by assigning them to a dummy platform $q_\emptyset$ and route $i_\emptyset$. This incurs a high penalty $c_\emptyset$ (e.g., 500).

Data available

Five JSON instances are available in which the size of the data set increases and makes the resolution challenging. The instances are

- A_small.json : 9 trains, 443 itineraries, 49 incompatibilities penalties and 0 authorization-related constrains
- inst_A.json : 46 trains, 443 itineraries, 2093 incompatibilities penalties and 0 authorization-related constrains
- inst_NS.json : 343 trains, 935 itineraries, 105632 incompatibilities penalties and 98 authorization-related constrains
- inst_PMP.json : 315 trains, 570 itineraries, 365996 incompatibilities penalties and 116 authorization-related constrains
- inst_PE.json : 265 trains, 2070 itineraries, 12152225 incompatibilities penalties and 0 authorization-related constrains

1 Reinforcment Learning modeling

1.1 State-space

We choose to consider the state-space as the list of trains, meaning that a state is a specific train which we will assign an itinerary. For the larger instance, the state-space dimension is 343.

1.2 Action-space

The action-space is the list of itineraries. A choosing action corresponds to the itinerary that will be assigned to a train. For the larger instance, the dimension of the action-space is 2070.

1.3 Transition probabilities

We decide to begin by taking the first id train, at each step we take the next train. While this procedure may limit the diversity of observed transitions and introduce a bias, we consider it acceptable as a first approach. Since the transitions are independent of the actions ($p(s', r|s, a) = p(s', r|s)$), we take the problem as a k-armed bandit with a dynamic environment.
At the beginning of each episode, all trains have unassigned platform tracks, the episode ends when each train has been assigned a platform or let be unassigned.

1.4 Managing large incompatible action-space

For each train, a large number of actions are incompatible, so we redefine the actions as a function of the state and the action space corresponding is the list of all compatible itineraries. In practice we define a dictionary

```
dict_it = {'train t' : [*list of compatible itineraries*]}
```

1.5 Reward function

We choose a large negative reward when the model takes an action that is unauthorized. And for each action taken, we take the penalties that this new itinerary introduces (cost c_j).

2 DQN Method

As an initial approach, we did not use the model described in Part 1. Instead, we considered a state-space representing the itineraries configuration, $[i_1,\ i_2, ..., i_T]$. However, this state-space had a dimension of approximately 10^{20}, even for the smallest instance. To address this, we applied function approximation and implemented the DQN method. Nevertheless, this model was far from optimal, so we revisited the original model to get the one described in Part 1. In this section, we present the DQN method applied to this smaller and more manageable model.

2.1 Implementation

We use the library Stable-Baselines3 to implement the DQN method. This method requires a gymnasium environment so we first construct the environment as described in Part 1 in the file Trains_gym.py and implement the following methods in order to get a gymnasium compatible environment : **init**, **_get_obs**, **_get_info**, **reset**, **step**, **close**, **set_itineraire**, **is_it_incompatible**, **is_quaie_interdit**. We choose a learning rate for neural networks of about $1e-3$, a buffer-size of about 800 and a discounted reward with $\gamma = 1$.

2.2 Results

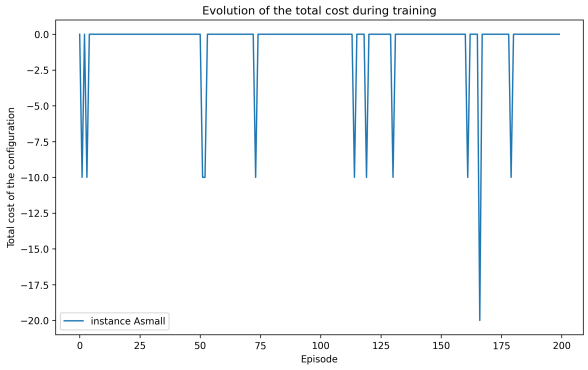

Figure 1: Instance A_small

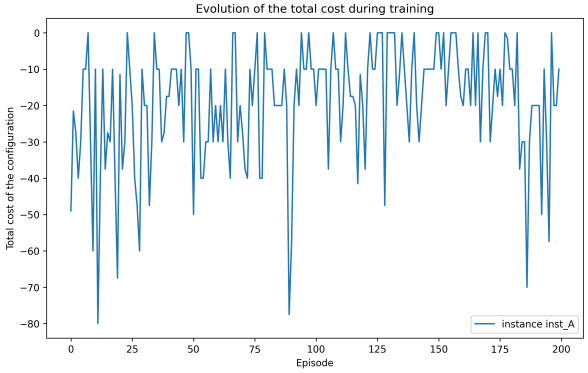

Figure 2: Instance inst_A

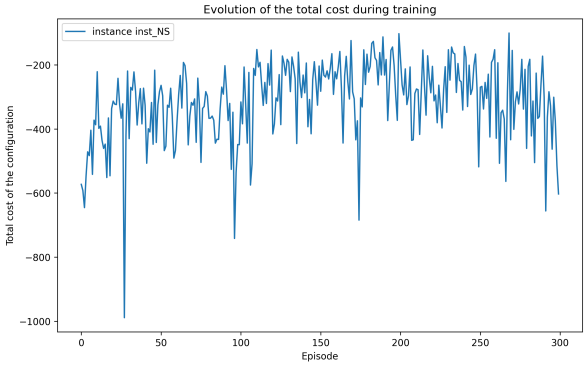

Figure 3: Instance inst_NS

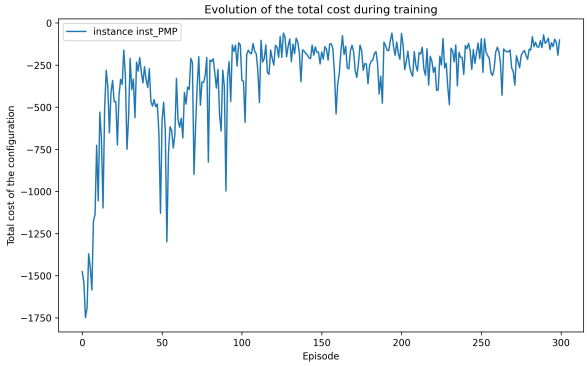

Figure 4: Instance inst_PMP

We plot the cost of the configurations at each episode, since we want to minimize it. We see that the simulations aren't stable and with larger instances, don't converge or don't do it fast enough. Yet, we can pick the configuration that produces the least cost. The results are given in Table 3. For instance in A_small, the optimal configuration assigns the itinerary 116 to the first train, 71 for the second, etc. This configuration gives no cost.
All the configurations can be seen in the notebook.

	Best cost	Best itineraries
A_small	0.0	[116, 71, 67, 202, 29, 337, 70, 64, 29]
inst_A	0.0	[93, 64, 366, 361, 374, ..., 110, 132, 329, 132, 329]
inst_NS	-100.66	[459, 905, 436, 436, 486, ..., 14, 874, 465, 14, 14]
inst_PMP	-30.0	[114, 221, 221, 219, 567, ..., 334, 511, 29, 306, 284]

Table 3: Best cost and configuration gotten from the simulations

This method is too unstable and with larger data sets doesn't converge in a few iterations. We decide to change the method and try the Tabular one.

3 Tabular MC Method

3.1 Implementation

For the implementation of the Tabular approach, we created a class *Environment* in the file **Trains-tabular.py** and implemented the following methods in order to perform a tabular version of the Monte-Carlo Algorithm. : **init , step , select action , contraintes itineraire , is quai interdit** and others detailed in the aforementioned file.

3.2 Results

On the following figures, we got pretty good results for all of the instances we had to work with. Contrary to the DQN method, the simulation converges and is quite stable. We can reach an optimal global assignation for each file and that the cost becomes consistently low as we iterate through the episodes. The instance int_PE is particular since the train 461 doesn't have any compatible platform track, then it is unassigned and the optimal cost cannot be 0. The evolution of inst_PMP is particular since it swings between unauthorized configuration (cost = 100) and authorized configuration (cost near 0). However, the model learns since the unauthorized configurations become rare with training.
We present Table 4 the optimal cost for each instance and the corresponding configurations are visible in the notebook.

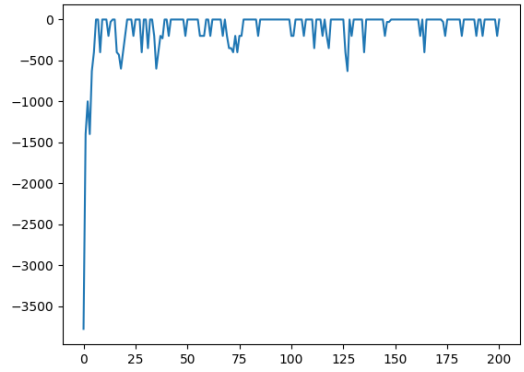

Figure 5: Instance inst_A

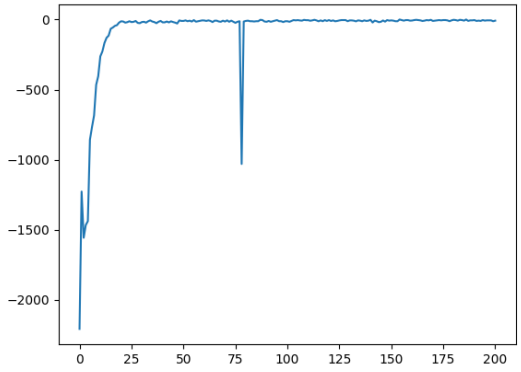

Figure 6: Instance inst_NS

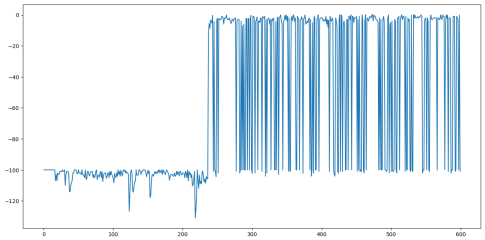

Figure 7: Instance inst_PMP

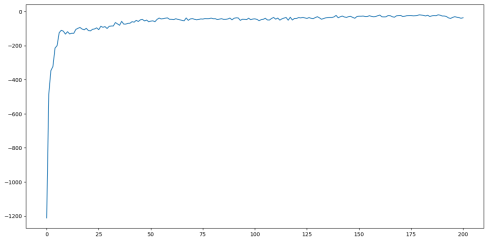

Figure 8: Instance inst_PE

	inst_A	inst_NS	inst_PMP	inst_PE
Best Cost	0	0	0	-70.76

Table 4: Best cost obtained during training

4 Outlook

4.1 Jax implementation

A way to speed up the computations and therefore have a usable model would be to implement our code using jax. We didn't do it because of a lack of time and because our mastery of the library isn't yet at the level where we were confident about implementing it without errors hard to debug.

4.2 Shuffle the list of trains

So far we treated the state space in the same order in every episode , meaning our start state was always the same for a given instance. While it still yields good results, a solution to increase exploration and reduce bias in the model would be to use exploring starts or, in other words, to shuffle the list if trains so as to treat it in a different order.