Reinforcement Learning
Elevator Dispatch

Students:
Gabriel Kasmi
Mehdi Bennaceur

Teachers:
Capgemini Team

May 17, 2020

Contents

1 Introduction

Modern societies are facing a wide variety of practical optimisation problems linked to our urban-modern lives, ranging from optimizing schedules to optimizing supply chain management for food stores. Traditionally ,this kind of problems was solved mainly through operations research techniques often incorporating problem-specific heuristics.

In this report, we will study a reinforcement learning approach to solve an Elevator control problem. In particular, we will summarize and re implement a method developed in (12). The elevator control problem consists in finding a way to control an elevator so that it *serves* [1] people while optimizing a certain criteria. The authors report that a criteria often used is close to a form of *average waiting time* which measures the average time an user will have to wait to get to his destination once he has pressed the button (calling the elevator). This is a very complicated problem which is often solved using heuristic based methods, although previous work attempting to provide a reinforcement learning based solution have been proposed in the literature ((2), (3), (6)). Algorithms used in practices in the industry are not well known since there are very rarely published for confidentiality reasons.

In a building, one can distinguish between 2 main scenarios which occurs during the day. the first one is in the morning where a lot of people are entering the building from the ground floor and need to go to some higher floor, this scenario is called the *up peak* scenario. The second scenario is the opposite, a lot of people arrive at various floors and all want to go to the ground floor, this is the *down peak* scenario. Usually, either one of the scenarios is studied in the literature instead of the entire problem (that is, a concatenation of the two with some mix of the two in between). It is often argued in the litterature, that the most difficult part is to handle these scenario because in the periods in between, fewer people are using the elevator and it is easier to serve everyone in a reasonnable amount of time.

The paper we will study focuses on the *down peak* scenario. A first part of the paper is dedicated to presenting the theoretical background necessary to understand the algorithms which are used in the following parts. The authors then implement and present the results achieved.

In this report, we will review the theoretical part and then present in detail the modelisation of the problem proposed by the authors. We then go on to discussing the implementation part and the experiments we have conducted. Finally, we will discuss the limitation of the model and make some critics.

[1]By serving we mean that the elevator answer the request of an user. That is, go to floor where the request was made and take the user to his destination

2 Theoretical Background

In this section, we summarize the main ideas in reinforcement learning and formulate the problem one wishes to solve. We then introduce the methods used to solve this problem. In section 2.1 we recall the framework and the main quantities we want to estimate, in section 2.2 we introduce the methods that have been historically used and are the foundations for the modern computational methods that we introduce in section 2.3. This section summarizes the main results and ideas from (Stutton and Barto), chapters 3 to 7.

2.1 Reminders on the reinforcement learning framework

In reinforcement learning, we consider an *agent* who interacts with an *environment*. The agent is the decision maker whereas the environment will only respond by sending feed-backs to the agent. Interactions between the agent and the environment are repeatedly occurring, so that the agent eventually *learns* his environment. The goal for him is to find the best behavior in order to maximize a given reward.

More formally, the environment will provide the agent some *state* S_t and a *reward* R_t. Based on these information, the agent takes an *action* A_t that will affect the environment who will in send another pair (state,reward) (S_{t+1}, R_{t+1}). Actions are taken within some set $\mathcal{A}(S_t)$ and rewards are assumed to be a scalar value $R_t \in \mathbb{R}$. We denote $\mathcal{R}$ the set of rewards (so by definition we have $\mathcal{R} \subseteq \mathbb{R}$) and $\mathcal{S}$ the set of states. The notion of "taking an action based on the environment feedback" is formalized by the notion of *policy*, denoted π and which is a mapping $\pi : \mathcal{S} \to \mathcal{A}$. This mapping can be either deterministic or stochastic, and we denote hereafter $\pi(a \mid s)$ the probability of taking action a given state s. The goal of reinforcement learning can be stated as follows : it consists in finding the best policy.

In order to determine which policy is the best, the agent evaluates the *rewards* it provides. Each action provides a reward and these rewards are used to compute the *return*, i.e. the sum of these rewards over the repeated interactions between the agent and the environment. We write :

$$G_t = \sum_{i=1}^{T} R_{t+i}$$

or if we consider an infinite horizon, $G_t = \sum_{i=1}^{\infty} \gamma^i R_{t+i}$ where $\gamma \in (0, 1]$ is a discount rate. On top of the time frame $t = 1, 2, \dots, T$ (with potentially $T = \infty$), we consider a second dimension, the episodes, corresponding to sequences were the game is played again. However, not all tasks can be considered *episodic.*

A crucial assumption on the environment is that the latter satisfies the Markov property, meaning that the dynamics of the environment can be described only resorting to the last time step's state and actions. A task that satisfies the Markov property is referred to as a Markov decision process (or MDP). MDPs can be either finite or infinite, depending on whether $|\mathcal{S}| < \infty$ or $|\mathcal{S}| = \infty$[2]. A MDP is entirely characterized by a state transition probability $p(s', r \mid s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)$.

Now that we've presented the main concepts and the most common framework considered in practice, let us introduce the value functions, which are the quantities one want to estimate in order to derive the optimal policy. We can consider either the value function of a state, denoted $v_\pi(s)$ or the value function or a (state, action) pair, denoted $q_\pi(s, a)$. These value functions are defined as the expected return under policy π. More precisely, we have:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i R_{t+i} \mid S_t = s\right] \\ q_\pi(s, a) &= \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i R_{t+i} \mid S_t = s, A_t = a\right] \end{aligned} \tag{1}$$

$v_\pi(s)$ is interpreted as the value of state s. It is called the *state-value function for policy* π. $q_\pi(s, a)$ gives the value of action a under state s and is called the *action-value function for policy* π. An important feature of these value function is that they can be estimated from experience. The optimal value functions are then the maximum over all possible policies of the state value function (resp. the action value function). It is possible for these optimal value function to derive Bellman optimality equations.

$$\begin{aligned} v_*(s) &= \max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_*(s')] \\ q_*(s, a) &= \sum_{s', r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{2}$$

For finite MDPs, these systems of equations can be solved and have a unique solution. From these solutions it is possible to derive an optimal policy π^* To do so, we use the fact that $\pi' \geq \pi \iff v_{\pi'} \geq v_\pi$. Note that the uniqueness of v_* does not imply the uniqueness of π^*. The goal is then to find computationnaly tractable methods for solving the system of equations given in (2).

[2]In the following, we focus on finite MDPs.

2.2 Foundations of optimal policy search

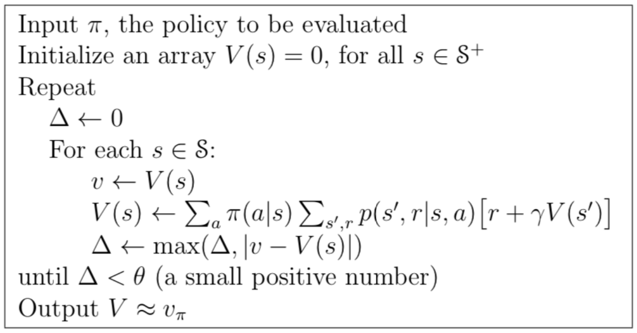
Historically, two methods have been proposed to compute the optimal policies: dynamic programming and Monte Carlo. These methods are the foundations of modern reinforcement learning methods that we will discuss in section 2.3.

2.2.1 Dynamic programming

Given a perfect model of the environment (i.e. assuming perfect knowledge of the latent MDP), it is possible to find v_* and q_* and thus to derive an optimal policy. The idea is to use the Bellman equations and to turn these equations into update rules. These equations are given in equation (3).

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}} \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r'} p(s', r \mid s, a)[r + \gamma v_*(s')] \\ q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{3}$$

The first idea introduce to solve the Bellman equations was to use policy evaluation, i.e. to compute the state value function v_π of any arbitrary policy. Given the expressions in equation (1), if the MDP is completely known, then we can solve this system of equations. However, a better idea is to use iterative methods, which are computationally more efficient. The policy evaluation algorithm is given in figure 1.



```
Input $\pi$, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$
```

Figure 1: Policy evaluation algorithm. Source : (Stutton and Barto)

We know that the value for changing of action a in state s and then following policy π' is given by $q_\pi(s, a)$. Since we want to find the best policy, we do not only want to evaluate the policies, but also to improve whenever possible. The **policy improvement theorem** ensures that a policy π' is better than a policy π as soon as for all $s \in \mathcal{S}$, $q_\pi(s, \pi'(s)) \geq v_\pi$. This result can be used to iteratively improve the policies, i.e. start from a policy π, improve it using v_π, get a better policy π' through policy improvement etc. In the case of a finite MDP, it can be shown that this process converges towards the optimal policy π^* (assuming the policy optimal policy is unique, otherwise one may switch between the two equally good policies). The idea is depicted in figure 2.

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       $a \leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
       If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ $false$
   If *policy-stable*, then stop and return $V$ and $\pi$; else go to 2
```

Figure 2: Policy iteration algorithm. Source : (Stutton and Barto)

Although policy iteration does converge to the optimal policy, it requires policy evaluation which needs to be done across all states. A way to get faster to the optimal policy (and not to wait until the limit is reached), thus reducing the computational burden of the method is to proceed via *value iteration.* This methods amounts to turn the Bellman optimality condition into an update rule.

More generally, the idea consisting sequentially improving and evaluating a policy can be referred to as generalized policy interaction or GPI. The improvement and evaluation steps (referred to as exploitation and exploration in the context of bandits) can be intertwined, such as for instance in greedy schemes. We will see that all reinforcement learning methods can be seen as GPIs.

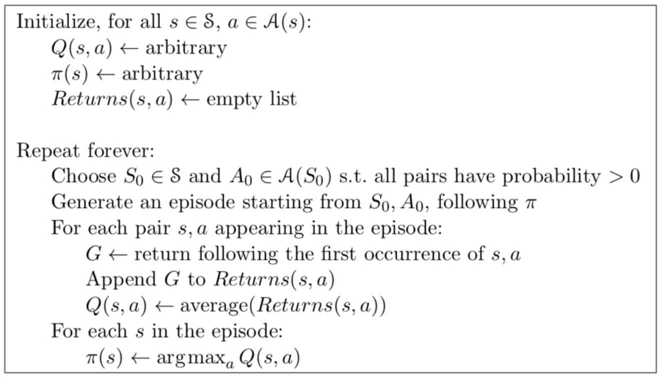
2.2.2 Monte carlo policy evaluation

So far, we've assumed perfect knowledge of the environment (i.e. of the latent MDP). With Monte Carlo methods, it is possible to derive the optimal policy without prior knowledge of the environment. However, value functions are computed over episodes averages, meaning that we can only apply these methods with episodic tasks.

The most straightforward Monte Carlo method used for policy evaluation is the first visit Monte Carlo, which consists in estimating the state value function by computing the average of the returns following the first visit to s with policy π. An alternative is the *all visits* Monte Carlo which averages these returns following each visit to state s. It can be shown that these methods converge to the true value function v_π (see (8)).

These methods can be extended to the estimation of state-action value functions $q_\pi(s, a)$. The only difficulty that arise in this case is that many (state,action) pairs may never be visited so one need to ensure sufficient exploration of the (state,action) space, otherwise this might hamper the improvement of the Monte Carlo estimates. A common assumption made is the *exploring starts*, according to which one will first explore all states.

As for the dynamic programming, we can apply Monte Carlo techniques in the GPI framework, i.e. alternating improvement and evaluation phases using Monte Carlo estimates of the state and state action value functions. We can then derive the strategy depicted in figure 3. This algorithm converges to the optimal policy.



```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s, a) \leftarrow$ arbitrary
    $\pi(s) \leftarrow$ arbitrary
    $Returns(s, a) \leftarrow$ empty list

Repeat forever:
    Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability $> 0$
    Generate an episode starting from $S_0, A_0$, following $\pi$
    For each pair $s, a$ appearing in the episode:
        $G \leftarrow$ return following the first occurrence of $s, a$
        Append $G$ to $Returns(s, a)$
        $Q(s, a) \leftarrow$ average($Returns(s, a)$)
    For each $s$ in the episode:
        $\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$
```

Figure 3: Monte Carlo exploring starts. Source : (Stutton and Barto)

Note that it is possible to tweak the algorithm above for a certain class of policies (ε-soft policies) such that the exploring starts assumptions is unnecessary for convergence. ε-greedy policies are an example of ε-soft policies.

The main advantages of Monte Carlo methods over dynamic programming are the following :

- They can be used to learn the optimal behavior directly from repeated interactions with the environment
- They can be used in simulation models
- It is possible to use them only on regions of the state space

Finally, these methods are less harmed by a violation of the Markov property.

2.3 Optimal policy search in the RL framework

Before turning to our case study, let us review the main results of the modern reinforcement learning theory. The first remark we can make is that the main difference between modern approaches and older ones lie in how the value function v_π is estimated. Methods for finding the optimal policy are then based on this estimation procedure and often resort on ε-greedy policies.

2.3.1 Temporal-difference learning and on-policy TD control

TD-learning can be seen as a combination of monte carlo and dynamic programming approaches. Recall that in the monte carlo setting, the value function is updated according to a formula close to the following rule :

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right]$$

This requires to wait until the end of the episode in order to know the value of G_t. The simplest TD method, called $TD(0)$ estimates G_t with the following quantity, which is immediately available, $R_{t+1} + \gamma V(S_{t+1})$. Thus the update rule becomes:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

and recalling the expression of $v_\pi(s)$, it turns out that the TD is an estimate for $v_\pi(s)$ both seen as $v_\pi(s) = \mathbb{E}(G_t \mid S_t = s)$ and $v_\pi(s) = \mathbb{E}(R_{t+1} + \gamma v_\pi(S_{t+1} \mid S_t = s)$. The main advantage of TD methods is that they can be implemented online, whereas Monte Carlo methods require to have a full episode available. Also, as compared to DP methods, they do not require knowledge of the environment.

A common method used to find the optimal policy when the $TD(0)$ approach is used to estimate the state value functions is the so-called SARSA method. Under TD(0), these methods converge toward the optimal policy as long as ε-greedy policies are considered.

SARSA is **on-policy** because it updates its Q-values using the Q-value of the next state s' and the current policy's action a'' It estimates the return for state-action pairs assuming the current policy continues to be followed. An alternative is to consider off-policy methods

2.3.2 Off-policy TD control : Q-learning

Q-learning is off-policy because it updates its Q-values using the Q-values of the next state s' and the greedy action a'. Put otherwise, it estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy. In its simplest form, the *one step* Q-learning update rule is the following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t\right]$$

We will see that in out paper, authors mainly present two RL algorithms which are the Q-value iteration and the Q-Learning algorithm. The main difference between the two is that the first one requires a *model* for the environment, in other words it needs to know a function f which describe the stochastic dynamics of the environment. The probability of ending up in state x_{t+1} as a result of action u_t in while state x_t is then $f(x_t, u_t, x_{t+1})$. The Q-value iteration algorithm is then more demanding than the Q-Learning one but can yield better results since it has more information.

In the implementation part the authors use a variant of the Q-Learning algorithm called $Q(\lambda)$ but it is not fully described. Since Q-value and Q-leaning are very similar algorithms we will here focus and describing the Q-learning and $Q(\lambda)$ algorithm. We decided to implement and test these two algorithm as discussed in the implementation part. The Q function we are trying to learning can be expressed as :

$$Q^\pi(s, a) = \mathbb{E}_\pi\left[R_t | s_t = s, a_t = a\right]$$

The fundamental results which makes this algorithm possible is the Bellman optimality equation which states that Q^* verifies :

$$Q^*(x, u) = \sum_{x' \in X} f(x, u, x')[\rho(x, u, x') + \gamma \max_{u' \in U} Q^*(x', u')]. \tag{4}$$

where $\rho(x, u, x')$ gives the reward obtained where transiting from state x to state x' by taking action u. The idea of Q-value iteration is basically to run a fixed point iteration of this last equation in order to converge towards Q^*. Since Q-Learning does not have access to the model of the environment through function f, the fixed point iteration equation it uses is the following one :

$$Q_{t+1}(x_t, u_t) = Q_t(x_t, u_t) + \alpha_t \cdot [r_{t+1} + \gamma \cdot \max_u Q_t(x_{t+1}, u) - Q_t(x_t, u_t)] \qquad \text{(Q-update)}$$

where α_t is a learning rate, γ the discount factor. We present the full algorithm below. The authors point out that convergence is theoretically guaranteed for this algorithm as long as :

- α_t is such that its sum is infinite but squared sum finite
- The agent keeps trying all actions in all states with non zero probability
- Explicit values for Q are stored and updated for each state and action.

The agent will explore the action state space by taking random action with probability ϵ and take an optimal action according to current Q estimates with probability $1 - \epsilon$. Each time updating the Q values according to the last equation. The full algorithm can be written as follows.

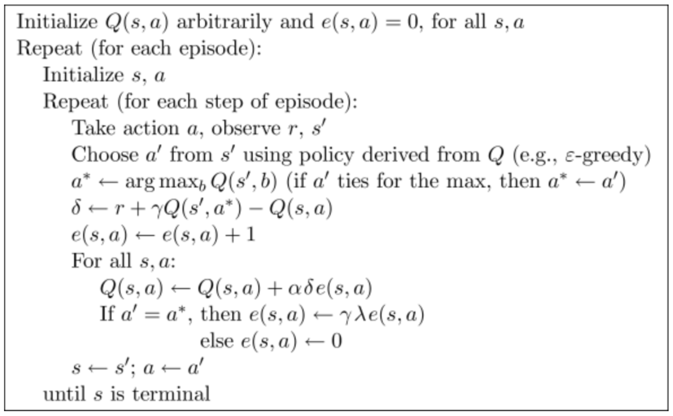
Algorithm 1 Q-Learning

```
Input: $\gamma, \alpha_t, \varepsilon, Q_0, T$
while $t \leq T$ do
    $u = rand()$ if $u \leq \varepsilon$ then
        Take random action $u_t$
    end
    Take greedy action according to $Q_t$
end
Observe new state $x_{t+1}$
Update $Q_{t+1}(x_t, u_t)$
```

We see that, at each iteration, we update one value of Q which correspond to the pair of state action in which the agent is before taking an action. The agent chooses an action, then observes the reward and the new states which makes it possible to use (Q-update). One can remark that when an action is taken, we only updates the last pair state, action, we might want to give credit also to previous state, action pairs encountered. This is the idea in a variant of Q Learning used by the authors and called $Q(\lambda)$.

2.3.3 $Q(\lambda)$ and eligibility traces

As expressed previously, in $Q(\lambda)$ we want to give credit to previous state, action pairs when an action is taken and an updating step taken. The parmaeter λ refers to the use of an eligibility trace, i.e. a temporary record of the occurence of an event. The algorithm presented here is basically the same as the Q-Learning except that it updates at each time step not only the Q-value of the last state-action pair, but also those of state-action pairs previously encountered, with weights that decay exponentially as the pairs go further back in time. The algorithm can be described as follows.



```
Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all $s, a$
Repeat (for each episode):
  Initialize $s$, $a$
  Repeat (for each step of episode):
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $a^* \leftarrow \arg\max_b Q(s',b)$ (if $a'$ ties for the max, then $a^* \leftarrow a'$)
    $\delta \leftarrow r + \gamma Q(s',a^*) - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all $s, a$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      If $a' = a^*$, then $e(s,a) \leftarrow \gamma\lambda e(s,a)$
                  else $e(s,a) \leftarrow 0$
    $s \leftarrow s'$; $a \leftarrow a'$
  until $s$ is terminal
```

Figure 4: $Q(\lambda)$ algorithm. Source : (Stutton and Barto)

We see in the algorithm that all state, action pairs are updated with a different weights as opposed to Q Learning which updates only one pair. In the paper, the authors claim faster convergence using this last algorithm. Before turning to the implementation part, we will present in section 3 the literature associated the problem we've studied as well as our own model of the system.

3 The elevator dispatching problem

In its most generic formulation, the problem is to minimize a given quantity (e.g. the waiting time) of elevator users of a given building, which comprises n elevators and k floors. This seemingly simple problem cannot be solved using dynamic programming methods since the cardinality of the set-space problem is generally huge. A solution has been to split the problem, i.e. to focus on specific moments, namely the morning when users go from the lobby to the higher floors (*up peak* scenario) or the evening when people leave their floor to go to the lobby (*down peak* scenario). But even in this case, depending on the formulation of the model, the state-space cardinality can still be to large for standard techniques to be efficient. In particular, dynamic programming techniques cannot be applied. It is the reason why prior to reinforcement learning, approaches based on heuristics were used in the industry.

The complexity of the problem was the main motivation behind (2) seminal paper. We will present their model, as well as further extensions proposed in the literature (section 3.1). We will then present our own approach, based on (12) in section 3.2. Our results are presented in section 4

3.1 Motivation and literature review

3.1.1 Conventionnal approaches

Let us first briefly review the approaches that were used before RL techniques were introduced. Algorithmic approaches to tackle elevator dispatching issues have been developped since the 1960s. The first approaches (e.g. (9)), using *collective control* consisted in stopping at the closest floor, with the disadvantage that several cars could reach the same floor. Research in the domain has also been hampered by the fact that systems are proprietary and descriptions designed for marketing rather than technical purposes. Nevertheless, numerous approaches have developped over the years:

- *zonal approaches* sur as for instance in (7) where each car controls a specific zone or area. This aims at keeping the elevators far from each other
- *search based approaches* in which a strategy is found by searching the space of possible car assignment and finding the one that minmizes a given criterion (e.g. waiting time). For instance in (13), when a button is pressed, the closest car is assignated.
- *rule based approaches* are approaches based on 'IF' 'THEN' statements. This approach was introduced by (11).
- *heuristic approaches* many heuristics approaches have been developped. These approaches aim at solving a specific situation, for instance trying to keep the cars evenly spaced (DLB approach) or assigning cars that are moving upwards to the longest queues (LGF).
- *adaptive and learning approaches* this is the last stream of methods, which use dynamic programming techniques. Due to the size of the state space, drastic simplifications have to be made. One of the first attempt to introduce dynamic programming for elevator

control has been proposed by (5). The RL methods can be seen as building upon the dynamic programming methods.

3.1.2 Multi agent RL to tackle elevator dispatching issues

(2) and (3) are to our knowledge the first attempt to tackle elevator dispatching problems using reinforcement learning. The elevator system they examine builds on earlier literature on this topic and considers a 10-story building with 4 elevators. The state-space dimension is enormous, and in such situations it can be interesting to consider several agents instead of a single one in order to explore the state space and find the best policy. In practice, each agent is responsible for one elevator car. The agents share the same objectives and cost functions and two architectures, namely parallel or decentralized are considered. The former correspond to one neural network for all agents whereas in the later each agent has its own neural network. With this article, a connection between the field of multi-agents reinforcement learning (see (1) or (4) for instance) and elevator dispatching problems.

The arrival of passengers at each floor is modelled by an inhomogeneous Poisson process (the rate of arrival varies in the course of the day). The autors focus on peak-down traffic, meaning that passengers arrive from floors 2 to 10 are are heading to the lobby. The physical constraints on the system taken into account are the floor, stop, turn and load times. The turn time corresponds to the time needed for a car to change direction. The load time follows a truncated Erland distribution with mean 1 second. Finally, each car's capacity is 20 persons.

Each car range of action is the following :

- Move up or move down if it is stopped at a floor
- Stop at the next floor or continue past the next floor if it is between floors.

Moreover, the followings constraints are taken into account:

1. A car cannot go past a floor if the user has requested this floor
2. It cannot turn until it has served all floors in its current direction
3. A car cannot stop at a floor unless requested (either to get on or off the floor)
4. A car cannot stop at a floor if another car has picked up passengers at this floor
5. If given the choice, a car would opt for going up

Constraint (5) was added in order to prevent the car from being "pushed-down" because of the peak down traffic.

For this kind of problem, there isn't really a natural target metric to minimize. Unsurprisingly, different metrics will yield different outcomes and policies. The most natural one can think of is the average wait time, but one could also consider the average system time (i.e. the sum of the wait and travel time). One could also want to minimize the share of passengers who have to wait more than a given amount of time (i.e. to minimize a *dissatisfaction* ratio. Finally, one can consider the squared wait times. The authors chose the latter metric and argue that this metric allows for low wait time and fair service. Since the setting we consider is continuous, one should rather use continuous instead of discrete returns. Indeed, the arrival times $t_1, \dots, t_T$ result from a continuous process. Formally, we write $G_t = \displaystyle\int_t^{\infty} \exp(-\beta(\tau - t)) r_\tau d\tau$ where r_τ is the instantaneous cost at time τ (so here it is the sum of the squared waiting time of all passengers).

The authors point out the fact that there are two possibilities for dealing with this problem : either consider all available information (*omniscient* learning) or only the information that would be available to a real operator (*online* learning) In the latter case, only the waiting time of the first passenger is known but it is possible to estimate the number of passengers in the queue waiting by reconstructing the intensity rate λ using the last inter-button time for that queue. Empirically, they noticed that the online approach yields satisfactory results.

Finally, the expected penalties during the first b seconds after the button is pressed is given by

$$\int_0^b \int_0^{b-\tau} \lambda \omega^2 e^{-\beta(\omega+\tau)} d\omega d\tau \tag{5}$$

A discretized version of the Q-learning algorithm is used with the following update rule :

$$\Delta\hat{Q}(x,a) = \alpha \left[\int_{t_x}^{t_y} e^{-\beta(\tau - t_x)} c_\tau d\tau + e^{-\beta(t_y - t_x)} \min_b \hat{Q}(y,b) - \hat{Q}(x,a) \right]$$

where action a is taken from state x at time t_x and the next decision is required for state y at time t_y. Solving for this equation in the Q-learning algorithm requires the knowledge of the waiting time of all passengers. As mentionned above, this difficulty is overcome by integrating by parts the expression in (5).

$Q(x,a)$ is estimated using neural networks, with 47 inputs and producing an estimate of the Q-value function as output. The 47 inputs have been found after experimentations and encode information about the hall buttons, the location and direction of the car, the floors where other cars are.

Another issue is that we do not know what is the optimal policy, so it is not possible to benchmark the performances of the proposed method. to overcome this difficulty, the authors compared their algorithm with heuristics algorithms used in the industry. Several cost functions were also considered and two variants of the RL controller were considered (decentralized and parallel). It turns out that both RL controllers were uniformly better than all heuristic algorithms for down peak profile with down peak only and generalized well to other situations, although trained on a down-peak profile only.

3.2 Modelling assumptions

Our implementation is based on (14). This article is a simplification of (2) and (3) since it only considers one car and a smaller state space. The advantage of this simplified approach is that it is possible to use standard RL techniques that would be intractable otherwise.

3.2.1 Elevator system

To model the elevator, the authors presented a simple model with fixed parameters which were chosen so that the state action space would be small enough in order to be able to apply the previously exposed algorithms. In our approach, we made some simplifications with respect to the autors' model. The following parameters are considered :

- The number of elevators, set to 1
- The number of floors, set to 5
- The height of a floor
- The elevator speed
- The elevator capacity, set to 4
- Stop time : time needed for passenger to exit and enter the elevator on a floor.

For simplicity, we decided to drop the speed and floor height variables (which are closely related). Dropping these means that we considered that the system will evolve in discrete time with a less precise approximation of a real system, we will define elementary actions which we will combine to model any sequence of action undertook by the elevator. From the point of view of the elevator, elementary actions are the following : moving up one floor, moving down one floor, waiting, taking people in, people exiting. We will consider that each of these actions takes up one unit of time. Therefore, any movement of the elevator in the building will be a sequence of these actions, and time will be easy to monitor. These assumption affects the dynamics of the elevator but it is still very close to what is done in the paper (as long as we don't compare the subsequent potential graphical interface to represent the system).

Furthermore, the authors assume a down peak scenario, which means that all requests will be from a floor to the ground floor. A request made by a user (i.e. someone pressing a button on a floor) is called a *call*. We assume that at most one passenger is waiting on each floor. Thus if one passenger arrives on a floor where there is already another waiting passenger, the number of waiting passengers on that floor is still viewed to be one. This is an important assumption which will make the implementation easier. The authors model the passenger to arrive at floors according to a stochastic process. In particular, an event e is drawn from the set $\{0, 1, 2, 3, 4\}$ with probability distribution $\{0.6875, 0.0625, 0.09375, 0.09375, 0.0625\}$. The event $e = 0$ means no passenger arrives, while an event $e > 0$ means a passenger arrives at floor $i = e$. We used a python class to model the elevator and how it interacts with the environment.

3.2.2 State and Actions

The state space originally lies in a 7-dimensional space, but here it will be in a 6-dimensional space since we dropped the speed parameter. A state $x = [c_1, c_2, c_3, c_4, p, o]$ is characterized by :

- c_i a binary variable indicating a call on floor i.
- $p \in \{0, 1, 2, 3, 4\}$ giving the position of the elevator.
- $o \in \{0, 1, 2, 3, 4\}$ giving the number of person in the elevator.

The resulting size of the state space $2^4 \cdot 5 \cdot 5 = 400$ which is small as intended by the authors. Note that by including the speed variable, the size increases to 1200. Now, the controller can chose between 3 actions: going up one floor $(+1)$, moving down one floor (-1) or wait (0). The action space is then $\{-1, 0, 1\}$, as stated before we will assume that each one of these actions takes one unit of time. In the paper, the same actions are used but the meaning slightly differ because the speed variable is taken into account. Indeed, the authors discretized the space between two floor into 10 small part and measure the position of the elevator using its speed. The actions used are accelerating upwards, stopping and accelerating downwards which is very close to our simplification.

The authors enforced some logical constraints on the choice of actions. The first one is that the controller cannot choose to go up $(+1)$ if the elevator is on the top floor, similarly it cannot choose -1 if the elevator is on the ground floor. The second one is that the elevator cannot switch direction instantaneously, that is action 0 must be taken between actions +1 and −1. We enforced the first constraint as well but the second is already verified by our system model since we didn't consider the speed variable and that our elevator discretely moves from floor to floor, it cannot change direction without stopping at one floor. However we decided to enforce another logical constraint which we believe will facilitate training. Since we are in a down peak scenario, if the elevator is at full capacity, then there is no point is considering any other actions than −1 until the elevator is on the ground floor and people exit the elevator.

3.2.3 Rewards and algorithm

We now turn to defining the rewards. It is first important to state the objective, what we would like the controller to achieve. The authors proposed two metrics which are often used for this problem in the literature. The first one is the *average waiting time*, an quantity computed as the average of the waiting time over all the passengers currently in the system. The second is the *trial waiting time*, which characterizes an entire run (trial) of the simulator. This is an average over all the time samples in the trial of the average waiting time defined above. When computing this quantity, it is assumed that the duration of the trial is finite. In this problem, we focus on the waiting time of passengers to evaluate the controller efficiency. With this objective in mind the authors define the following reward :

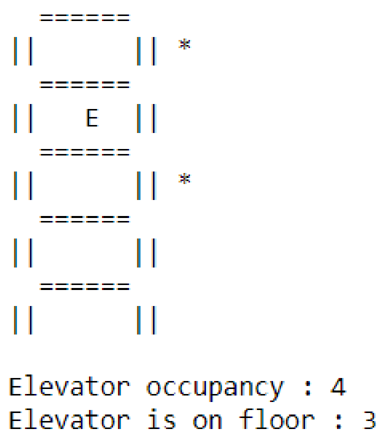
$$r(x) = -\sum_{i=1}^{4} c_i - o$$

In other words, at state x the reward is the negative sum of the occupancy of the elevator and the number of calls. This indeed measure the number of people waiting and we see that the reward is maximized only in the case where no one is waiting.

Finally, the updates of the Q-value function is done according to (Q-update) in order to provide an estimate for (4). We use the Q-value iteration algorithm.

4 Implementation and Experiments

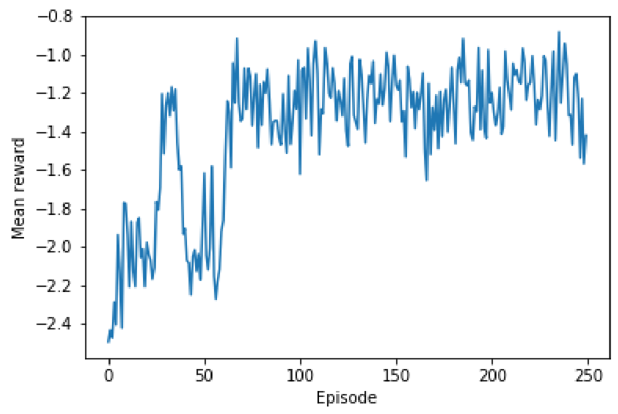
Before implementing an algorithm to solve the problem, we first need to implement an environment which will simulate the elevator and passengers while monitoring the time elapsed. To do that, we created a simple python object which also allows to run simulations (Figure 5). This last point is useful for debugging purposes and allows for a visual and intuitive evaluation of policies. On the figure the E represents the position of the elevator and the stars represents calls, legend in the bottom monitor the occupancy of the elevator as well as the floor its located at. This was useful during our experiments to see how the elevator behaved and detect anomalies.



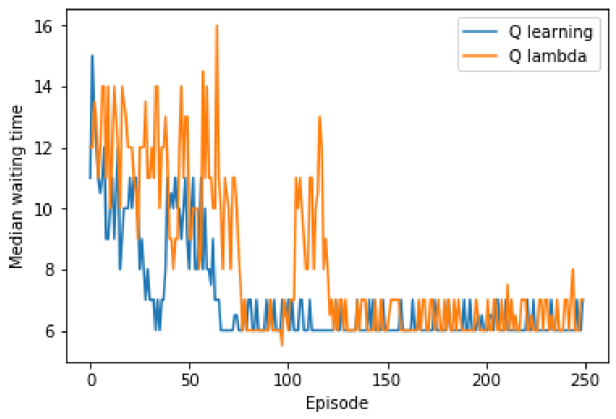
```
  ======
||      || *
  ======
||   E  ||
  ======
||      || *
  ======
||      ||
  ======
||      ||

Elevator occupancy : 4
Elevator is on floor : 3
```

Figure 5: Simulation interface

We implemented a simple Q-Learning algorithm and a $Q(\lambda)$ as discussed in the previous part. For both we ran several episodes, one episode consisting in 500 unit of time. We found similar performances for both methods (Figure 6) although the $Q(\lambda)$ method takes much more time to run but this is certainly partly due to our implementation. We see that the agent goes towards earning a mean reward around -1 (Figure 6a) (it is roughly the same for both methods). We also computed the mean waiting time and see that both perform similarly. Given that the algorithm are still exploring even after several episodes, we decided to plot the median waiting time instead of the mean waiting time which is more sensible to outliers.



(a) Mean Rewards for Q Learning



(b) Median Waiting Time

Figure 6: Results of Q Learning and Q(λ)

The time is not necessarily meaningful in absolute value because of the assumptions we have made, that is that each elementary action takes one unit of time so that moving up one floor takes the same amount of time as 4 peoples exiting the elevator.

However, we see can interpret some aspect of the results. We achieve a median waiting time of around 6 units of time, which means half of the passengers are served in less than 6 units of time. By looking at the distributions of calls, we see that half of the passengers arrive on floor 1 or 2. Now if the elevator is on floor 0 and a call is made on floor 2, it will take at least 6 units of time to serve the passengers (if the elevator was already on floor, it would take at least 4 units of time). So the performance of the algorithm seems acceptable. Furthermore, by executing a simulation (using the *implement_policy* function in the notebook) one can judge of the quality of the policy.

Finally, we have also benchmarked the Q learning method with a naive approach. It consists in operating the elevator in a deterministic way which makes it go from the ground floor to the top floor, going back down and so on. The elevator is going through all the floor stopping only to load and unload passengers and is otherwise constantly moving. This yields an average waiting time around 9.5 and the same for the median. The gap between this naive policy and the one found by Q-Learning would certainly increase as the number of floors increases, or if we changed the capacity of the elevator.

5 Discussions

We have reviewed and implemented our version of the algorithm presented in (12) which aims at providing an elevator control framework. We were also able to visually inspect the quality of the policy obtained using a simpler animation than the one presented in the paper. We believe that this formulation constitutes a good introduction to the problem of elevator control and its challenges even though it is still too simple to be applied in a real life setting.

Indeed, the discrete modelisation used here does not fully describe the real motions of an elevator and the subsequent constraints one can imagine (acceleration, deceleration). Furthermore, a more realistic example would likely be closer to a continuous setting and the use of simple Q-Learning would not be possible simply because the storage of state, actions pairs would not be possible or even don't make sense. As mentioned in the paper, a real life setting would also be likely to include multi agent framework in which each elevator would be controlled by an agent.

References

[1] Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In *Advances in neural information processing systems*, pages 295–302.

[2] Crites, R. H. and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In *Advances in neural information processing systems*, pages 1017–1023.

[3] Crites, R. H. and Barto, A. G. (1998). Elevator group control using multiple reinforcement learning agents. *Machine learning*, 33(2-3):235–262.

[4] Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In *Advances in neural information processing systems*, pages 271–278.

[5] Levy, D., Yadin, M., and Alexandrovitz, A. (1977). Optimal control of elevators. *International Journal of Systems Science*, 8(3):301–320.

[6] Makaitis, D. (2003). Evolving fuzzy controllers through evolutionary programming. In *22nd International Conference of the North American Fuzzy Information Processing Society, NAFIPS 2003*, pages 50–54. IEEE.

[7] Sakai, Y. and Kurosawa, K. (1984). Development of elevators supervisory group control system with artificial intelligence. *Hitachi review*, 33(1):25–30.

[8] Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158.

[9] Strakosch, G. (1967). *Vertical transportation: elevators and escalators*. Wiley.

[Stutton and Barto] Stutton, R. and Barto, G. Reinforcement learning: an introduction. 1998.

[11] Ujihara, H. and Tsuji, S. (1988). The revolutionary ai-2100 elevator-group control system and the new intelligent option series. *Mitsubishi Electric Advance*, 44.

[12] Xu Yuan, Lucian Busoniu, R. B. (2008). Reinforcement learning for elevator control.

[13] Yoneda, K., Sakai, Y., Matsumaru, H., Tobita, T., and Yasunobu, S. (1991). Elevator control system. US Patent 5,042,620.

[14] Yuan, X., Buşoniu, L., and Babuška, R. (2008). Reinforcement learning for elevator control. *IFAC Proceedings Volumes*, 41(2):2212–2217.