UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DEVELOPMENT OF A REINFORCEMENT LEARNING MODULE BASED ON Q-LEARNING OF A MULTI-AGENT SYSTEM FOR EMERGENCY EVACUATION SIMULATION

GUZMÁN GÓMEZ PÉREZ

2019

TRABAJO DE FIN DE GRADO

Título:	Desarrollo de un módulo de Aprendizaje por Refuerzo basado en Q-Learning de un Sistema Multiagente para Simulación de Evacuación de Emergencia
Título (inglés):	Development of a Reinforcement Learning Module based on Q-Learning of a Multi-agent System for Emergency Evacuation Simulation
Autor:	Guzmán Gómez Pérez
Tutor:	Carlos A. Iglesias Fernandez
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO DE FIN DE GRADO

Development of a Reinforcement Learning Module based on Q-Learning of a Multi-agent System for Emergency Evacuation Simulation

Guzmán Gómez Pérez

Enero 2019

Resumen

Este proyecto se propone como una toma de contacto entre dos áreas en auge y con un enorme potencial por delante: el *Machine Learning* y la Simulación Multiagente. En concreto, esta memoria expone la integración del algoritmo *Q-learning* (dentro del subgrupo del Aprendizaje por Refuerzo) como técnica utilizada para la evacuación de un incendio, y llevada a cabo en una plataforma multiagente basada en la simulación de situaciones de emergencia dentro de edificios.

Q-learning lleva al mundo de la programación la ecuación de Bellman, que consiste en la ponderación de un conjunto de intentos ejecutados como prueba y error dentro de un problema de optimización, y cuyo resultado se procesa en forma de valor númerico. Esta combinación permite al algoritmo interiorizar el funcionamiento de cualquier contexto acotado con normas, patrones y estados, alcanzado un desempeño óptimo en una fracción del tiempo que le llevaría a un ser humano. Como se demostrará en esta memoria, el *Q-learning* desbanca al algoritmo *Astar* en su función de buscador de ruta de evacuación óptima gracias a su alta eficiencia computacional. Además, en su implementación se ha fomentado la flexibilidad y facilidad de adaptación a cualquier problemática, para poderlo reciclar con multitud de aplicaciones diferentes.

Dado que se trabaja sobre una plataforma orientada a la investigación y al desarrollo constante, la implementación de este algoritmo se ha llevado a cabo integramente, a través de Python, sin librerías externas de Inteligencia Artificial que abstraigan partes fundamentales del código. Esto ha permitido una mejor integración del algoritmo, personalizando sus características para sacar el mayor partido de las mismas. Ofreciendo una monitorización del proceso más completa, con diversas fuente de información en forma de logs y representaciones gráficas. Que aportan incluso un cariz didáctico a la plataforma.

Palabras clave: Q-Learning, Simulación Social basada en agentes, evacuación, MESA.

Abstract

This project is proposed as a contact between two booming areas and with enormous potential ahead: Machine Learning and Multi-agent Simulation. Specifically, this graduate thesis presents the integration of the Q-learning algorithm (within the subgroup of Reinforcement Learning) as a technique used to evacuate a fire, and carried out in a multi-agent platform based on the simulation of emergency situations inside buildings.

Q-learning takes the Bellman equation to the world of programming, which consists of the weighting of a set of attempts executed as trial and error within an optimization problem, and whose result is processed in the form of a numeric value. This combination allows the algorithm to internalize the functioning of any context bounded by standards, patterns and states, achieving optimal performance in a fraction of the time it would take a human being. As it is going to be demonstrated in this project, the Q-learning overrides the A star algorithm in its function of optimal evacuation route finder thanks to its high computational efficiency. In addition, its implementation has fostered flexibility and ease of adaptation to any problem, so that it can be recycled with a multitude of different applications.

Given that it works on a platform oriented to research and constant development, the implementation of this algorithm has been carried out entirely, through Python, without external libraries of Artificial Intelligence that abstract fundamental parts of the code. This has allowed a better integration of the algorithm, personalizing its features to get the most out of them. Offering a more complete monitoring of the process, with various sources of information in the form of logs and graphic representations. Additionally, this gives a didactic utility to the platform.

Keywords: Q-Learning, Agent-based Social Simulation, evacuation, MESA.

Agradecimientos

Gracias a mi madre, a mi hermana y a Laura por hacerme ver siempre la mejor versión de mi mismo.

Gracias a Carlos Angel Iglesias por reafirmar y potenciar mi interés en el área del *Machine Learning*.

Gracias a todos los que habéis contribuido con vuestro granito de arena a la consecución de mi grado.

Contents

Resumen **VII**

Abstract **IX**

Agradecimientos **XI**

Contents **XIII**

List of Figures **XV**

1 Introduction **1**

1.1 Context 1

1.2 Project goals 2

1.3 Structure of this document 2

2 State of the art **5**

2.1 Agent-based model 5

2.1.1 Agent-based social simulation 5

2.1.2 MESA 6

2.1.3 SOBA 6

2.1.4 SEBA 6

2.2 Machine Learning 6

2.3 Reinforcement Learning 7

2.3.1 Dynamic Programming: Value iteration 10

2.4 Q-learning 11

2.4.1 Exploration vs Exploitation Policy 12

2.4.1.1 E-greedy. 12

3 Requirement Analysis **13**

3.1 Introduction 13

3.2 Use case diagram 13

3.3 Use cases 15

4 Architecture **19**
4.1 Introduction . . . 19
4.2 Classes overview . . . 19
4.2.1 State class . . . 21
4.2.2 Model class . . . 23
4.2.2.1 '*grid*' and '*grid_resources*' parameters . . . 23
4.2.2.2 State instance . . . 25
4.2.2.3 '*n_episodes*' and '*n_episode_steps*' parameters . . . 25
4.2.2.4 Multi-agent approach . . . 28
4.2.2.5 QLearning instance . . . 29
4.2.3 QLearning class . . . 31
4.2.4 Occupant class . . . 37

5 Case study **39**
5.1 Introduction . . . 39
5.2 Application of Q-learning to a dynamic fire evacuation . . . 39
5.2.1 Viability analysis . . . 39
5.2.2 Logistic implementation . . . 41
5.2.3 Simulation results . . . 43
5.3 Sensibility analysis . . . 44
5.3.1 Comparison among evacuation strategies in SEBA . . . 44
5.3.1.1 A star versus Q-Learning . . . 45

6 Conclusions and future work **49**
6.1 Conclusions . . . 49
6.2 Achieved goals . . . 49
6.3 Future work . . . 50

A Impact of this project **51**
A.1 Introduction . . . 51
A.2 Description of relevant impacts related to the project . . . 52
A.3 Conclusion . . . 52

B Economic budget **53**

C Neural Networks and Q-learning **55**

Bibliography **58**

List of Figures

2.1 Context for Q-Learning and Agent-based Model fields. 7
2.2 Sample of a generic RL Markov Decision Process [2]. 8
2.3 RL general structure [12]. 8
2.4 Visualization of the converging process in Value iteration [1]. 11

3.1 UML Use Case diagram. 14

4.1 UML Class Diagram of the project's implementations. 20
4.2 State-Value visual abstraction concept. 22
4.3 RAMEN's and MESA's visualization perspectives. 23
4.4 String and numeric abstractions of the example's environment. 24
4.5 Number of episodes, optimality and number of alternatives correlation. . . . 26
4.6 Number of episodes, optimality and number of alternatives correlation. . . . 27
4.7 Multi-agent approach convergence. 28
4.8 Multi-agent Q(s,a) sequential expansion along the grid. 29
4.9 SSD of the overall learning procedure in Model class. 30
4.10 SSD of the overall inferring procedure in Model class. 31
4.11 SSD1: in-depth learning procedure of QLearning class. 33
4.12 Unicode most optimal action per cell. 34
4.13 SSD2: in-depth inferring procedure of QLearning class. 35
4.14 SEBA's visualization of three agent QLearning fire evacuation cases 36
4.15 Verbose mode printed representation of the followed path. 37

5.1 Example of the generalization of all possible fire combinations. 40
5.2 Agent grid migration process. 42
5.3 Activity diagram of the implemented evacuation procedure. 42
5.4 Simulation 1 of a dynamic fire evacuation with Q Learning. 43
5.5 Simulation 2 of a dynamic fire evacuation with Q Learning. 44
5.6 Big-O notation computing complexity 46
5.7 Computing time per strategy scaled by the number of agents 46
5.8 Average deaths per simulation with a variable number of agents 46

5.9 Survival rate as alive persons from total number of persons. 47

C.1 DeepMind's Deep Q-learning sctructure applied to a SEBA's sample input image. 56

CHAPTER 1

Introduction

This chapter includes a summarized explanation of the context concerning this project, the consequent challenge that this implies, and the approach that this project will follow to solve it.

1.1 Context

Emergency situations inside buildings occur more and more frequently. Shopping centres, airports, offices, educational institutions, etc. These are places where the population spends most of their time. A correct execution of the evacuation within these constructions is essential to avoid the multiple potential tragedies.

The use of multi-agent simulation systems is nowadays, the most accurate way to approximate the behaviour of large social masses, anticipating valuable conclusions that save costs and accelerate the research [16].

This project arises as a study proposal that continues with the research initiated by the end of degree project: Design and Implementation of an Agent-based Crowd Simulation Model for Evacuation of University Buildings [8]. The idea is to insert the Q-learning algorithm in the evacuation process, by swapping it for the current used method: A-Star algorithm [6]. The heavy computational processing of this optimal mathematical distance

algorithm implies great limitations when escalating the environment.

In contrary, the nature of the multi-agent systems fits perfectly with reinforcement learning techniques. The idea is to dynamically mine information from micro-interactions between agents and the environment, while process it by using the built-in algorithm Q-learning. The resulted data feeds back the behaviour of these agents in such a way that they optimize their actions based on previously defined guidelines.

In this project, the multi-agent framework is performed by three overlapping platforms working as a whole: MESA [10] (generic multi-agent simulation), SOBA [7] (moves this simulation inside buildings), SEBA [8] (SOBA extension that incorporates the modelling of emergency situations, in particular of fires).

The algorithm will be applied into static and dynamic fire distribution scenarios. The results will be measured by evaluating the terms of survival, execution time, computational complexity and implementation reusability.

1.2 Project goals

- Implement the Q-learning mechanism priorizing: flexibility, reusability and transparency. So it could be easily reused for other applications such us energy efficiency, connectivity and more.
- Improve survival rate. This can be achieved by the joint action of the following points.
- Drastic reduction of the time execution in order to avoid the current 'freezes' when computing escaping routes, by taking advantage of the better computational efficiency of the Q-learning algorithm.
- With A-star it is only possible to apply one strategy per simulation. Q-learning could efficiently combine all current escaping strategies: less crowded path, the safest path and shortest path.
- Evaluate the possibility of adding additional knowledge to the Q-learning process related to affiliation psychology.

1.3 Structure of this document

This section lists a brief overview of the chapters which structure this document:

1.- Introduction: Summarized explanation of the context concerning this project, as well as the carried out approach.

2.- State of the art: Thorough description of all main standards and technologies involved in the provided resolution.

3.- Requirements: UML user-system interaction description of the software developed in this project.

4.- Architecture: Description of the design details involving the applied implementations with static fire distribution.

5.- Case Study: Extension of the architecture procedure into dynamic fire learning case.

6.- Conclusion: The most relevant information deduced from the obtained results as well as a proposed approach to apply in future works.

CHAPTER 2

State of the art

This chapter includes an introduction of the main concepts concerning this project.

2.1 Agent-based model

The aim of **agent-based modelling** (ABM) [5] is to simulate big scale networks in an attempt to anticipate unexplored situations from which to extract useful conclusions. The applied methodology consists on establishing a set of simple rules to define the interaction between autonomous and heterogeneous agents in a non-trivial way according the characteristics of the environment. Randomness is often integrated as a representation of the agent criteria through several ways such as Monte Carlo or stochastic environments. The application concerning this project is social sciences, specifically the **Agent-based social simulation** (ABSS) [11].

2.1.1 Agent-based social simulation

The ABSS technique is a branch of ABM that emerged as the joint work of three fields: social science, agent-based computing and computer simulation. Its utility consists on integrating agent technology for simulating social micro and macro phenomena on a computer in real time. One of the most extended frameworks used for purpose is **MESA** [10].

2.1.2 MESA

MESA provides a versatile framework for building, analysing and visualizing agent-based mode. It allows users to quickly create agent-based models using built-in core components (such as spatial grids and agent schedulers) or customized implementations, visualize them using a browser-based interface and analyse their results using Python's data analysis tools.

2.1.3 SOBA

This framework is an extension of MESA created as a Simulator of Occupancy Based on Agents inside buildings and implemented on Python. As described in SOBA's documentation [7]: 'The simulations are configured by declaring one or more types of occupants, with specific and definable behaviour and a physical space. Regarding space, two different models are provided: a simplified model with a room defined by rooms, and a model with a continuous space. The simulation and results can be evaluated both in real time and post-simulation'.

2.1.4 SEBA

SEBA [8] (Simulation of Evacuations Based on SOBA) is a simulation tool for studies related to emergencies and evacuations in buildings. It shares the same modules of SOBA as it is an extension of it. The difference lies in the addition of another agent profile module which represents the fire behaviour. It is also necessary to include the interaction rules between persons agents and the fire agent. The modules where the code integration will take place are the ones included within the classes Occupant and Model.

2.2 Machine Learning

The most extended understanding about software's functionality has been based on the execution of instructions that a programmer had previously defined in order to reproduce a desired behaviour. The importance of artificial intelligence (AI) [14] lies in avoiding the role of the intermediary, the programmer, by enabling the software to learn and decide based on its own experience. One of the main branches that derives from AI is **Machine Learning** (ML) [13].

ML encompasses a wide variety of methods which can be grouped in four main distributions: **Supervised Learning**, **Unsupervised Learning**, **Semi-supervised Learning**

and **Reinforcement Learning**. Depending on the details of the problem to solve, one approach will fit better than the others.

This thesis focuses in the Reinforcement learning area, specifically on the Q-learning algorithm, where the learning procedure is carried out by accumulating rewards gathered as a result of the interaction between an agent (RL algorithm) and the context.

The Fig. 2.1 shows a visualization of the hierarchy of the most relevant technical fields of this project.

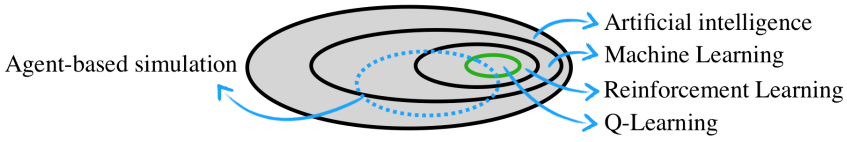

Figure 2.1: Context for Q-Learning and Agent-based Model fields.

2.3 Reinforcement Learning

The first thing that should come to mind when thinking about Reinforcement Learning (RL) [12] is a context with two main parts: the **agent** and the **environment**. Until both elements are explained, let's imagine a black box that defines the relationship between them.

The agent represents the algorithm, that is, an entity with cognitive abilities (memory and deduction). Identifying the agent is simple, it is the element that owns conduct (defined by a certain set of actions) and on which the optimal execution of the problem to be solved depends.

Nevertheless, the environment is always structured as sequential sets of alternatives from which to infer an optimal combination. These alternatives are shaped as states. The environment also owns information related to every single state, which must at least contain an associated reward. The environment has to fit any model which follows a **Markov Decision Process** (MDP), as shown at Fig. 2.2. Additionally, the agent must be given a certain initial state from which to start the transition process. It ends when the final state is reached or interrupted by an external event.

Let's now discover the inside of the black box. First the environment receives an external status and hands it to the agent. Then the agent provides a certain **action** (denoted as $\boldsymbol{a}$) to the environment and computes its next state ($\boldsymbol{s}$). The suitability of that action (in that state) is measured by the next state's associated **reward** ($\boldsymbol{r}$), weighted among the previous chosen state-action pairs and stored in a variable through the Q-learning algorithm. The

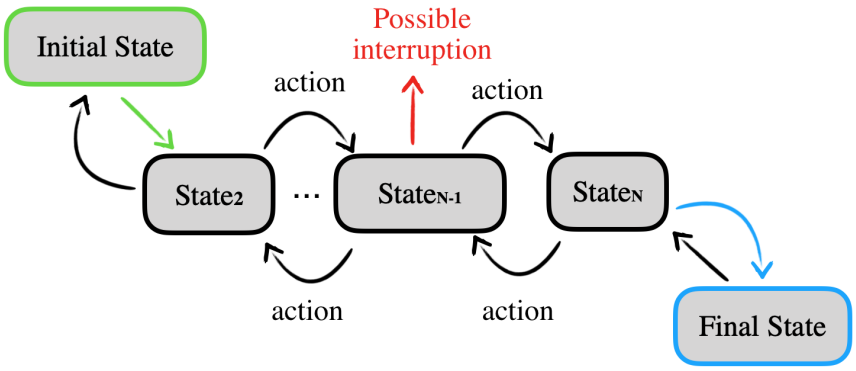

Figure 2.2: Sample of a generic RL Markov Decision Process [2].

process is repeated until reaching a ending state or external event. An overview of the process is shown at 2.3.

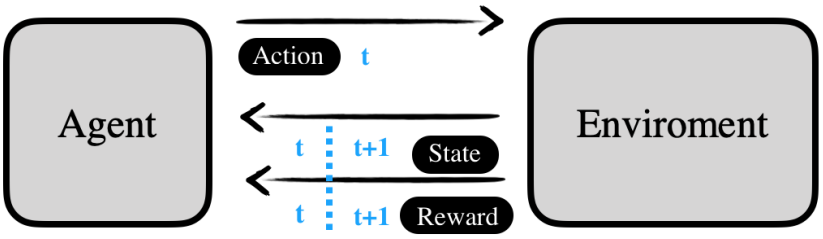

Figure 2.3: RL general structure [12].

In other words, the action is the trigger that leads to the next state. That is to say, the transition between states will depend both on the current state and on the actions allowed. Therefore, the transition will be represented as $\boldsymbol{p(s_{t+1}|s_t, a_t)}$.
The choice of an action is given by a defined strategy referred to as **policy**, denoted as $\boldsymbol{\pi}$. It can be imagined has a function where the input is the current state and the output is the next action to take, see eq. 2.1. That is, it defines the behaviour our agent is expected to have, and varies according to the case to deal with.

$$\boldsymbol{a = \pi(s)} \tag{2.1}$$

The policy can be either **deterministic** or **stochastic**. The first term refers to a case where there is no place for randomness in the choice of the next action (and the state it leads to). The second one alludes to a range of actions where its choice depends on probability. The optimal policy $\boldsymbol{\pi^*}$ is the sequence of actions that leads to a resolution of the process in the optimal way. That is, the one that maximizes the total reward.

The reward $\boldsymbol{r}$ traduces the influence of each of the states where the agent steps into. This value is represented by a number, and its size is proportional to the harm or the benefit that a certain state means to the agent. Defining these values is as simple as listing all the possible states and give them a numeric score (reward) proportional to the suitability of the situation.
The total reward or accumulated experience mentioned above is called **cumulative reward** as $\boldsymbol{R}$. In order to optimize the learning process, it is common to apply a **discount factor** in such a way that the experience gathered in the first steps has greater weight. In other words, the importance of the rewards that the agent finds along its way decreases according to the amount of states reached since the start. This factor is represented as $\boldsymbol{\gamma}$ and its values are included within the interval $\mathbf{0} \leq \boldsymbol{\gamma} \leq \mathbf{1}$.

$$\boldsymbol{R} = \sum_{\boldsymbol{t=0}}^{\boldsymbol{T}} \boldsymbol{\gamma^t r_t} \tag{2.2}$$

The **state-value function**, denoted as $\boldsymbol{V(s)}$, measures the suitability of a certain state regarding the total reward the agent has collected until reaching that position, that is the cumulative reward from the initial state to the input state. The path followed depends on the applied policy.

$$\boldsymbol{V^{\pi}(s)} = \mathbb{E}[\sum_{\boldsymbol{t=0}}^{\boldsymbol{T}} \boldsymbol{\gamma^t r_t}] \quad \forall \boldsymbol{s} \in \mathbb{S} \tag{2.3}$$

The specific case where the path is given by the optimal policy $\boldsymbol{\pi^*}$ is shown below.

$$\boldsymbol{V^*(s)} = \max_{\boldsymbol{\pi}} \boldsymbol{V^{\pi}(s)} \quad \forall \boldsymbol{s} \in \mathbb{S} \tag{2.4}$$

It can be useful to observe this formula reversed to understand the contrary point of view, in which the optimal policy comes from the sequence of states that maximizes the state-value function.

$$\boldsymbol{\pi^*} = \boldsymbol{arg} \max_{\boldsymbol{\pi}} \boldsymbol{V^{\pi}(s)} \quad \forall \boldsymbol{s} \in \mathbb{S} \tag{2.5}$$

Another necessary function to describe in the RL algorithms is called **state-action value function**. It goes one step further by designating a suitability score to each of the enabled actions in one particular state. This function is alternatively called Q-function $\boldsymbol{Q(s, a)}$ and its output is known as **quality value**, as it measures the quality or convenience that a certain action implies to the agent in its state when solving the problem. It is equal to the summation of the inmediate state reward and the discounted optimal state-value function of the next state $\boldsymbol{s'}$, where the input action leads to.

$$\mathbb{E} : \mathbb{S}\boldsymbol{x}\mathbb{A} \rightarrow \mathbb{R} \tag{2.6}$$

$$\boldsymbol{Q(s,a) = R(s) + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s')} \quad \forall \boldsymbol{s} \in \mathbb{S} \tag{2.7}$$

In the same way as the state-value function, the resulting accumulated reward varies depending on the selected policy. As before, the state-action can maximize the action-value by choosing the optimal policy. This case is denoted as $\boldsymbol{Q^*(s,a)}$.
It is possible to combine both value functions resulting in the equivalence:

$$\boldsymbol{V^*(s) = \max_a Q(s,a)} \quad \forall \boldsymbol{s} \in \mathbb{S} \tag{2.8}$$

Therefore, $\boldsymbol{V(s)}$ function can be expressed as a recursive call of the discounted state-value function of the next state plus the sum of the reward in the current state. This formula is popularly known as **Bellman equation**, in honour to its inventor, the mathematician Richard Bellman.

$$\boldsymbol{V^*(s) = \max_a [R(s,a) + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s')]} \quad \forall \boldsymbol{s} \in \mathbb{S} \tag{2.9}$$

In order to optimize the learning process, Bellman's equation uses the **Dynamic Programming** method [4], enabling the possibility of applying two main variants: **Policy iteration** and **Value iteration**. The aim of both methods is to converge to the optimal policy. The one applied in this project is the Value iteration policy.

2.3.1 Dynamic Programming: Value iteration

The dynamic programming optimize complex problems that can be discretized and sequenced. This technique fits perfectly the resolution of an optimization problem based on a Markov Decision Processes. As the information of the environment is distributed in independent states, the challenge is to progressively optimize the weight among the sequence of visited states as the agent keeps on exploring the environment.

In Fig. 2.4 the green arrow represents a stochastic policy, that is, a random selection of state-action pairs along the path. The blue one represents a policy that always applies a deterministic criteria. The convergence is accomplished by comparing policies dynamically while creating the path. That is, the random and deterministic criteria used to chose each state-action pair alternates (black arrows).
The deterministic policy improves its criteria along the process advances, thanks to the weighted state rewards. Then, the construction of the optimal policy is immediate once

all possible combinations of $\boldsymbol{Q(s,a)}$ have converged to an accurate representative value by selecting the maximum $\boldsymbol{Q(s,a)}$ in each state. The understanding of these concepts will be reinforced in the next section: Q-learning.

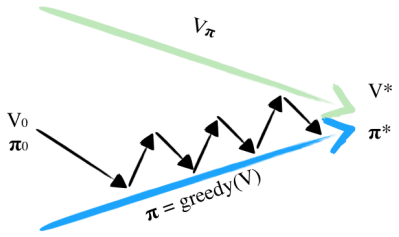

Figure 2.4: Visualization of the converging process in Value iteration [1].

2.4 Q-learning

Q-learning is a type of Reinforcement Learning which uses **model-free learning**. That is, the agent starts blinded, without having knowledge about the reward distribution nor defined transitions between states. Then he infers an optimal policy by interacting with the context. Under these conditions, Q-learning uses the **Temporal Difference Learning** method in order to learn the $\boldsymbol{Q(s,a)}$ values of the environment by sampling the different state-action pairs Q function with a variable degree of randomness.

$$\boldsymbol{Q_n(s,a) = (1-\alpha)Q_p(s,a) + \alpha Q_o(s,a)} \tag{2.10}$$

$$\textit{where} \quad \boldsymbol{Q_o(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a')} \tag{2.11}$$

Before starting the explanation of equation 2.10 let's clarify the used notation **n**, **p** and **o** subsymbol's mean as new, previous and observed respectively. To begin with, all the $\boldsymbol{Q(s,a)}$ pairs are initialized to the same neutral value in order to build a fair and accurate quality value distribution of the environment. The left side of the equation represents the pair state-action Q function $\boldsymbol{Q_n(s,a)}$ that is going to be updated. This value is equal to the summation of the previous stored same state-action Q function pair $\boldsymbol{Q_p(s,a)}$ and the observation of the next state's $\boldsymbol{Q_o(s,a)}$. In order to optimize the learning, $\boldsymbol{Q_p(s,a)}$ will be given more weight as the agent accumulates more experience. For this purpose it will be multiplied by the $\boldsymbol{1-\alpha}$ factor, where $\boldsymbol{\alpha}$ is called learning rate. In contrary, $\boldsymbol{Q_o(s,a)}$ will have less relevance as the agent knows more about the environment. After each sample finishes, $\boldsymbol{Q_n(s,a)}$ happens to be $\boldsymbol{Q_p(s,a)}$ and the process is repeated several times until the $\boldsymbol{Q(s,a)}$ of that pair state-action converges to the value which most truly represent its suitability.

2.4.1 Exploration vs Exploitation Policy

This technique is easily explained with an ordinary simile. Let's imagine a man lost in the jungle. In order to survive the man decides to explore his surroundings in search of feeding resources. As the man walks through the jungle and generates experience, he learns how to avoid dangers while successfully finding food. As a rational being, he starts to exploit that knowledge, that is, he utilizes the optimal policy in order to maximize the environment's reward. To sum up:

- **Exploration**: random choice of any of all the possible actions $\boldsymbol{a}$ in state $\boldsymbol{s}$.
- **Exploitation**: choice of maximum $\boldsymbol{Q(s,a)}$ from all possible actions $\boldsymbol{a}$ in state $\boldsymbol{s}$.

There are several strategies to distribute the occurrence proportion between these two events. The choice among them depends on the problematic dealt with. The most commonly used and one of the most efficient of them is the one applied in this project: **E-greedy** [15].

2.4.1.1 E-greedy.

In this approach, the epsilon $\boldsymbol{\epsilon}$ factor behaves similarly to the learning rate. At the first steps, the agent barely knows what to expect from the environment because of the lack of experience. But unlike $\boldsymbol{\alpha}$, $\boldsymbol{\epsilon}$ acts as a probability. Its magnitude is directly proportional to the occurrence of $\boldsymbol{Q(s,a)}$ exploitation's method and consequently inversely proportional to its complementary event: $\boldsymbol{Q(s,a)}$ exploration. In each state's transition a random probability sample $\in \boldsymbol{[0,1]}$ determines whether the factor $\boldsymbol{1-\epsilon}$ tilts the balance towards the exploration as a random choice among actions or towards the exploitation, $\boldsymbol{\epsilon}$, by choosing the maximum expected $\boldsymbol{Q(s,a)}$ pair. As the agent learns and gets more confident, epsilon decreases.
The most commonly applied probability distribution starts at $\boldsymbol{\epsilon = 0.99}$ and at the start of each episode it is reduced by a factor of $\boldsymbol{0.99}$ divided by the number of trials (episodes), until epsilon reaches $\boldsymbol{0.1}$. This non-zero range is necessary in case of non-static environments where the exploration must never end, in order to keep adapting to certain possible changes.

CHAPTER 3

Requirement Analysis

3.1 Introduction

This section exposes all the possible interactions between the user and the system chronologically presented through use cases.

3.2 Use case diagram

The use cases can be divided into three main parts: primary actors, secondary actors and the system. All the involved actors are detailed in table 3.2. As MESA is the underlying framework of SOBA, and SEBA is built as an extension of it, along the process SEBA represents the three as a whole. The entire interaction through the system can be seen at 3.1, the sequence is ordered from top to bottom. First, the simulation user designs a building scheme through RAMEN's framework [3], generating a file with that information. Then, SEBA translates that data into implemented variables, shaping the simulation's building structure. Those variables are extracted by this project's implementation in order to model the building environment. Afterwards, the algorithm is prepared in two phases: the parameter tuning and the learning process. Once generated the 'q-table', the real time fire simulation can be performed. Its visualization is optional, if demanded, it can be reproduced either 2D through SEBA or 3D with RAMEN.

Table 3.2: Primary and secondary actors

Actor identifier	Name	Type	Role	Description
ACT-1	User	Primary	User	The user that wants to solve an optimization problem in a building simulation context using the Q-learning technique.
ACT-2	SOBA	Secondary	Framework	The framework which supports the building simulation scheme.
ACT-3	SEBA	Secondary	Framework	The framework which supports the building evacuation simulation for emergency simulations like fire.
ACT-4	RAMEN	Secondary	Framework	A framework which renders a 3D simulation of a given '.json' file. In this case, shows the resulted SEBA-SOBA simulation in a 3D view.

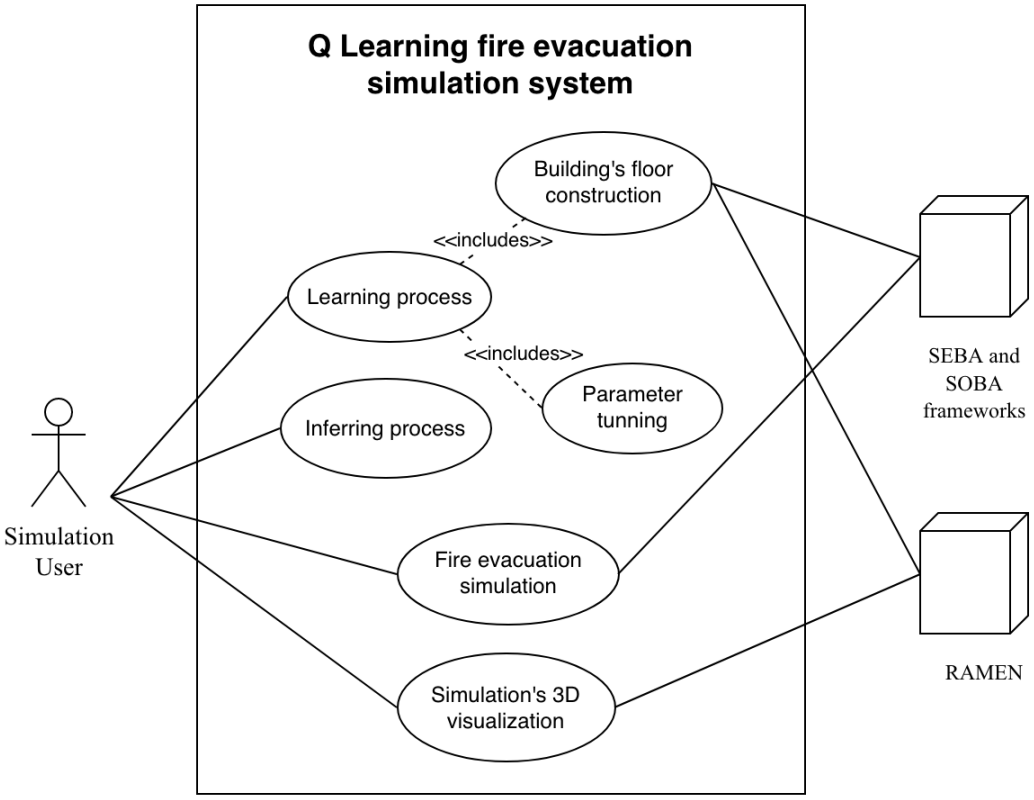

Figure 3.1: UML Use Case diagram.

3.3 Use cases

The uses cases are sequentially listed below with its respective overall analysis and a reference pointing to an schematic table. Every table starts by specifying its use case name and identification number. Afterwards, it follows the pre-condition field, describing the necessary events that must or should precede that use case. Finally, the flow events is composed by a row per event, listed in chronological order from top to bottom. In each row, the actor input encompasses all the necessary actions made to achieve the system response (following column). The first three uses cases share an experimental and preparatory purpose. Its code is implemented in a Jupyter Notebook file, supported by logs and plots, that extracts from SEBA the necessary objects to describe de building (environment) design. The last use case, uses the learned stored data to apply the evacuation at SEBA's real time simulation. All the technical details involving the algorithm will be explained in-depth in the architecture described in section 4.

1. **Parameter tuning (table 3.4)**: Each environment (building) has a different number of state-action combinations (obstacles, dimensions, etc.), therefore, the algorithm's parameters must be adjusted in order to achieve an efficient algorithm resolution.

2. **Learning process (table 3.6)**: The Q-learning algorithm splits the procedure into two parts; first the learning and then the path inference. This use case describes the first part: configuration and execution of the algorithms state-actions quality values convergence.

3. **Inferring process (table 3.8)**: This use case describes how to enable and execute the second part of the Q-learning algorithm: optimal solution inference (best evacuation path) through maximizing the final state value function from the learned data.

4. **Fire evacuation simulation (table 3.10)**: This use case assumes the algorithm is ready (the learned data must be specified in pickle format at a given file) and explains the procedure to apply the Q-learning algorithm in parallel with SEBA's simulation to evacuate the agents from fire through the shortest and safest path. The visualization can be seen in 2D or 3D, or directly runned in batch mode without any rendering.

Table 3.4: Use case to adjust the algorithm to a certain building floor.

Use Case	Parameter tuning	
Use Case ID	UC1	
Pre-Condition	First, SEBA must have provided the String grid abstraction of the building.	
Flow Events	Actor Input	System Response
1a	The user accesses to the 'Q-learning_sensibility_analysis' Jupyter Notebook file and follows the detailed steps to find the parameter values which best fits in the given design.	An approximation of the best 'number of episodes' and 'number of steps per episode' values.

Table 3.6: Use case of the Q-learning first part: training the agent.

Use Case	Learning process	
Use Case ID	UC2	
Pre-Condition	The UC1 should have been done in order to obtain a more accurate result.	
Flow Events	Actor Input	System Response
1a	The user specifies the 'number of episodes' and 'number of steps' parameters according his criteria (computing preferences, tuning results, etc.).	The learning process starts, the needed time interval varies depending on the building floor size and the number of posible actions. When finished, the 'q_table' is generated and stored in the 'q_learning_files' folder in pickle format. If specified, the solution is visualized.
1b	The user can optionally add or modify the starting coordinates from where the episodes will train the agent. It is recommended to pick the corners of the floor.	
1c	The user can optionally add any further concrete situation to the fire evacuation simulation by accessing to the 'grid resources' built parameter. Such as: modifying the reward priorities, adding new grid reward-symbol pairs, etc.	
2	The user creates a Q-learning instance to call the 'learn' method. Optionally, the user can render several visualization perspectives of the learning process.	

Table 3.8: Use case of the second part: optimal resolution with 'q-table' data.

Use Case	Inferring process	
Use Case ID	UC3	
Pre-Condition	The UC2 must have been previously done. Therefore, the 'q_table' should be in the 'q_learning_files' inside 'auxiliarFiles' SEBA's folder.	
Flow Events	Actor Input	System Response
1a	The user specifies the 'number of steps' parameter, which must coincide with the used in the learning process (UC2), of that envrionment.	The system looks for the correspondent saved 'q_table'. Translates it from pickle to object. Instantly returns the most optimal solution (route) and the visualization representations if they were asked for.
1b	The user specifies the 'grid' and 'grid_resources' parameters. These must be the same as the used in the learning process (UC2), of that environment.	
1c	The user defines a start state from where to compute the inferring of the optimal solution.	
2	The user creates a Q-learning instance with those parameters and runs the 'infer_path' function. Optionally, there can be also runned multiple functions to visualize this solution from different perspectives.	

Table 3.10: Use case to apply Q-learning evacuation strategy in SEBA's framework.

Use Case	Fire evacuation simulation	
Use Case ID	UC4	
Pre-Condition	UC1, UC2 and UC3 preparation must be done.	
Flow Events	Actor Input	System Response
1a	The user changes the 'update_q_learning_files' variable into False, and runs the SEBA's 'run.py' configuration file.	Saves the 'q_table' of the current .blueprint3d grid in pickle format.
1b	The user configures the 'run.py' configuration file to customize the simulation.	Stores the specified parameters such as number of agents, social roles, etc.
1c	The user changes the 'update_q_learning_files' variable into False and executes the 'run.py' configuration file in batch mode (-b) or MESA's visualization mode (-v).	Execution and (optional) rendering of the simulation. The agents follow the infered route in order to evacuate the floor.

CHAPTER 4

Architecture

4.1 Introduction

In this chapter the design phase of this project is covered, as well as implementation details involving its architecture. Firstly, it can be found an overview of the project, divided into several brief explanations of the functionality of each class separately and as a whole. After that, each of them is presented in much more depth.
In order to reinforce the comprehension, the used example to support all generic Q-learning applications (through SOBA) will be a simplified version of the one approached in this project. It consists on a SEBA's evacuation simulation in an environment with static fire. That is, all fire positions remain where started as it does not expand.

4.2 Classes overview

The figure 4.1 illustrates the class diagram of this project, it integrates the implementation added in both frameworks (SOBA and SEBA). Keep in mind that in the diagram there has not been included all attributes and functions belonging to these classes, instead, in order to focus the explanation on the relevant aspects, there has only been specified the ones with a direct intervention in the Q-learning process approached in this project. To extense this in-

formation see the framework's documentation referenced at 2.1.3 (SOBA) and 2.1.4 (SEBA).

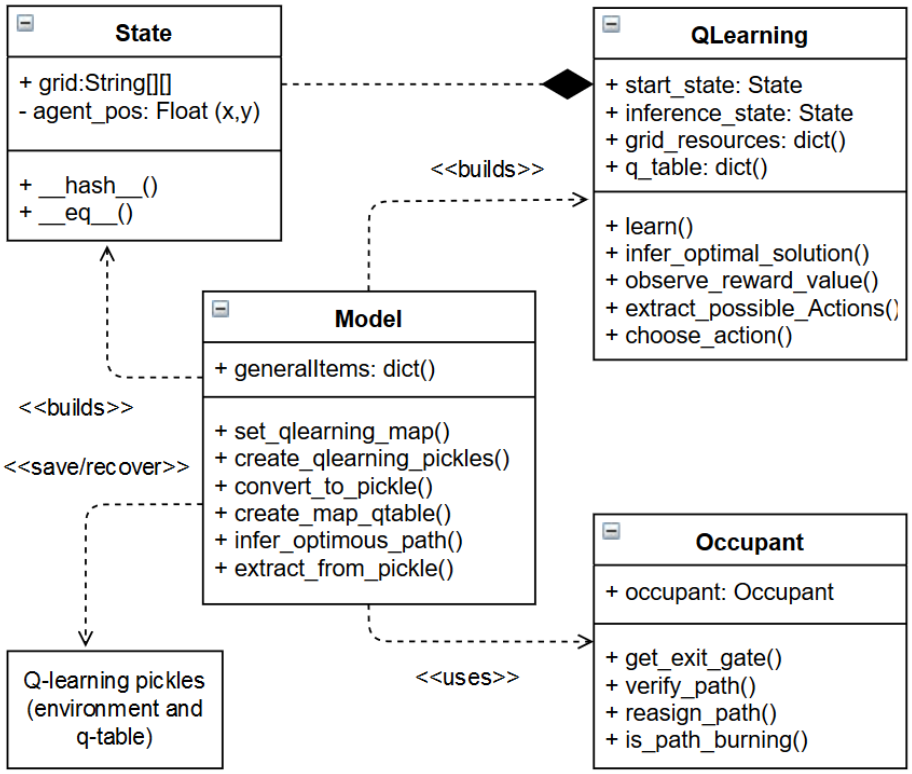

Figure 4.1: UML Class Diagram of the project's implementations.

The **QLearning** and **State** classes belong to the '***e_greedy_q_learning***' module located in SOBA's framework. The State class represents a Markov Decision Process state (explained in State of art's subsection: 2.3) and includes all the information related to it. The algorithm is developed in the QLearning class, which uses the State class to define each of the states of the environment that the agent transits through.

The **Model** and **Occupant** classes belong to the SEBA framework. As explained in its section, the Model class creates the building simulation model on emergency situations. Therefore, it collects all the simulation information and orchestrates its interaction. The main used elements are: space distribution (rooms, doors, walls, obstacles, etc.) and agents distribution, whether social roles or fire application. Afterwards, in the real time execution the Model's step function will coordinate its interaction supported by the time module.

When applying the Q-learning evacuation, the Model class builds the Markov Decision Process environment as a two-dimension *string* list spatially distributed and calls the

learning and inferring functions provided by QLearning's instance. Moreover, along the simulation, the Occupant asks the Model for the inferred optimal path according to each agent current position, in case the emergency mode starts due to fire detection. As this process happens in real time simulation, and the learning process is much slower, it is necessary to generate the *q_table* before the execution. Thereafter, the results are saved in a separate file (in pickle format), enabling an instantaneous resolution of the learned optimization problem, computed in milliseconds.

Henceforth, these four classes will be explained in depth level.

4.2.1 State class

To obtain a depth understanding of this class it is necessary to orientate the explanation towards the ***q_table*** dictionary of the Qlearning class. As mentioned in section 2.3, the aim of the learning process is to obtain the state-action quality values. The *q_table*'s code looks like the following:

```
q_table = {State (param1, param2, ...) : [q1, q2, q3, q4, q5, q6, q7, q8],
    ...} .
```

As it can be seen, the State's instance acts as the 'key' of the *q_table*'s dictionary and its input parameters gather all the specific information that characterises a certain state. In the implemented case, these parameters are: the 2D grid that models the environment and the state's position as a two-coordinates tuple *(x, y)*. Anyway, these parameters could be changed or extended depending on each optimization problem requirements. This univocal information will be used to locate back that state when the agent steps on it again. In this way, that State's instance will weigh all experienced related to itself in each learning session.

As this class is mainly used to search a certain match among all the possible environment states, it is very important to implement it with hashed properties. This fact will dramatically speed up the learning process saving up to ten times the original timing. This justifies the ***__hash__()*** function, but why is ***__eq__()*** added? Although the hash classification range is approximately 64 bits, there is still a very small possibility that two state objects have the same hash.

To continue with, the 'value' of the dictionary, related to each of the mentioned state's keys, is a one dimension list of length equal to the number of actions that can be carried out in any single state. Inside, a numerical *float* type value represents the suitability of

each action. To go deeper over this concept rewind to section 2.3. Anyhow, figure 4.2 will reinforce its understanding by showing a visual abstraction of this concept. The figure must be complemented with the notes listed below:

- The *(1,1)* state has been selected by random criteria in order to expand its information, but this can be extrapolated to the rest of states.
- The displayed arrows represent the actions of the SEBA's evacuation optimization problem.
- The order in which the quality values are shown is again generic, as in practice this ranking would obey to the suitability of each state's action.
- The environment is defined by the gridded 2D list.

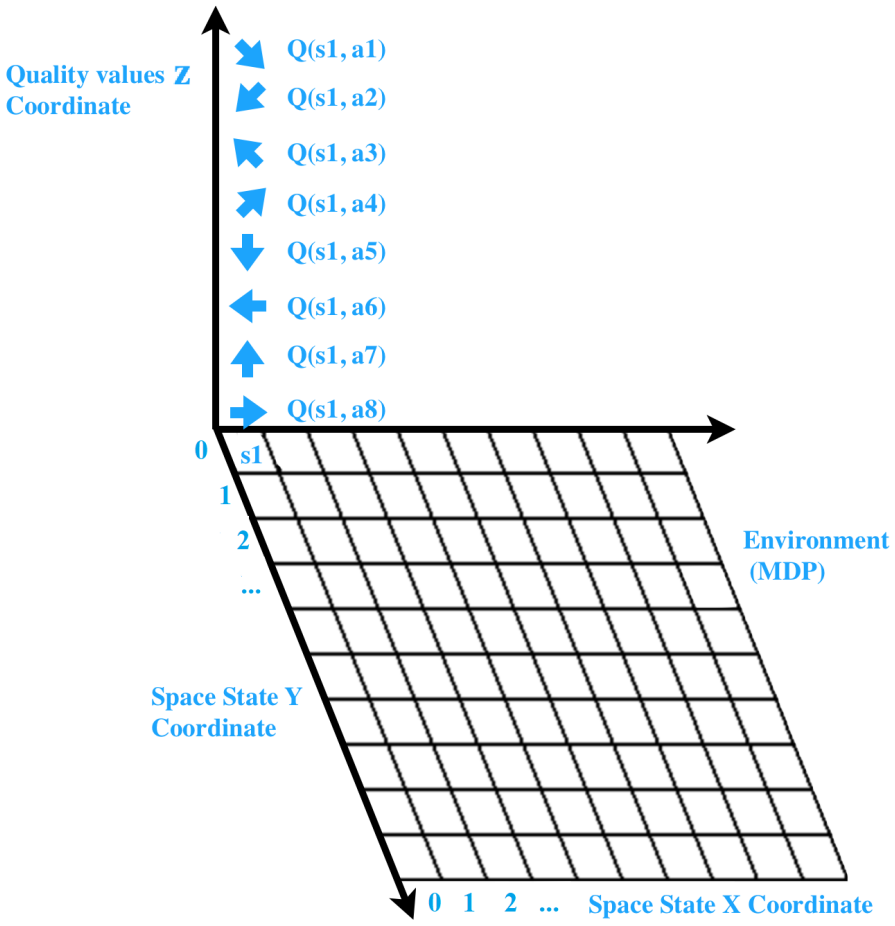

Figure 4.2: State-Value visual abstraction concept.

4.2.2 Model class

This class creates the necessary SEBA's application parameters and then handles them to the Qlearning instance. Although both, the inferring and learning function, share the same parameters, their values might vary depending on the applied one.

4.2.2.1 'grid' and 'grid_resources' parameters

In order to understand the role of this class in the Q-learning application, it will be necessary to delve into the creation of the environment. Figure 4.3 and 4.4 show all its possible perspectives on the case of SEBA's evacuation simulation model.

Let's begin explaining the left side's simulation showed in figure 4.3 as it is the most realistic perspective among the four shown. This 3D visualization of the model is the result of the RAMEN framework [3].
The right side picture reproduces the same building structure from above, in a more minimalist way. This 2D rendering has been obtained as the simulation execution from the MESA's visualization implemented tool on SEBA's platform. In this example, it is easy to associate colour grey with obstacles, the green with interest points (places were a certain activity is carried out) as well as exit doors, red square with fire and the blue circle with an agent (person).

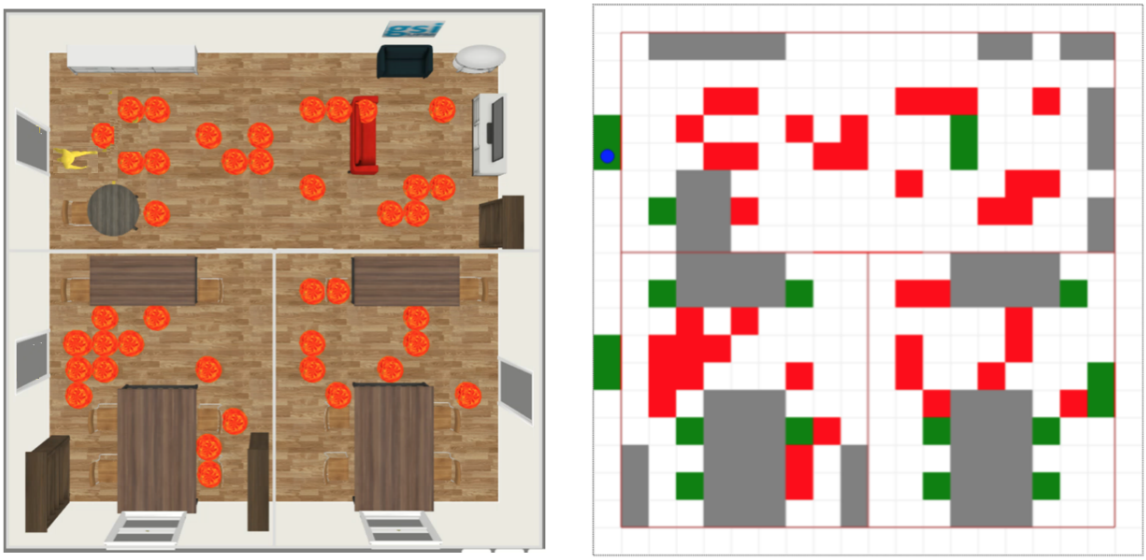

Figure 4.3: RAMEN's and MESA's visualization perspectives.

Let's now focus on figure 4.4. The right picture represents the ***grid*** parameter as the

standard 2D *string* list it has been talked about in previous paragraphs. Each cell contains a character that represents an element of the simulated model. In this example 'f' refers to fire, '.' means empty space, 'w' represents the wall, 'd' symbolizes an inner door and 'x' an exit door. As said in theory 2.3, when the agent steps on any of these characters, the impact on experience is measured as a reward proportional to its benefit or harm.
The numeric grid on the left side provides the reward equivalence as:

- Fire → -10: as it is the highest negative value, the agent will prioritize avoiding these states.
- Exit door → +100: as it is the final goal to achieve it has the highest reward of all grid's elements.
- Empty spaces and inner doors → -1: to penalize distance and force the agent to infer the shortest path. If distance was irrelevant, the reward should be neutral (0).

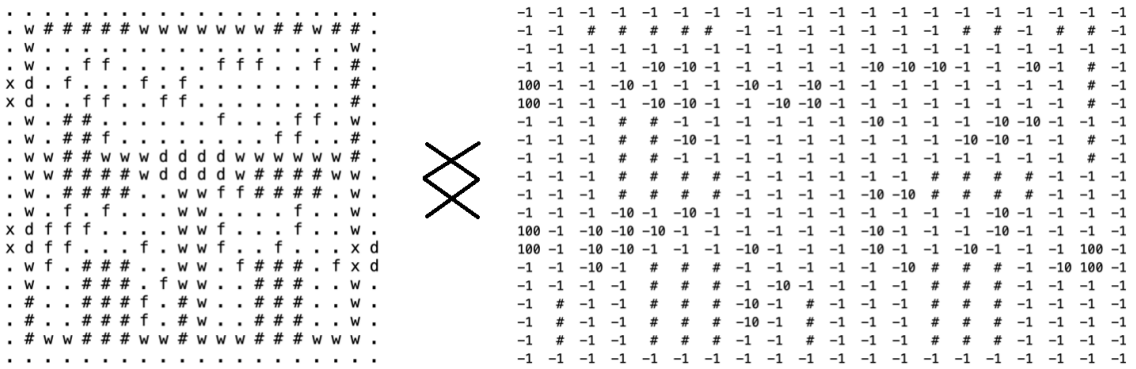

Figure 4.4: String and numeric abstractions of the example's environment.

The ***grid_resources*** parameter contains the information of each *string* symbol of the grid, enabling the Qlearning class to interpret it. Additionally, it can be added any additional object with further utility along the Qlearning process, like any information concerning the environment that cannot be described as an string-reward pair. The symbols with an associated reward must strictly follow this structure:

```
{string_symbol: [associated_reward, ends_the_episode], ..., otherObject}
```

- ***string_symbol***: The *string* of the symbol to which the information of the value's list is associated.

- ***associated_reward***: The reward as a *float*, being positive if suitable or negative if avoidable.

- ***ends_the_episode***: *Boolean* type; *True* if stepping into this symbol (state) ends the episode, *False* otherwise.

The example below continues with the SEBA's evacuation case, showed in figure 4.4, note that the obstacle symbol '#' is just used to identify and avoid those states:

```
grid_resources = {'.': [-1, False], 'w': [-1, False],  'd': [-1, False], 'f
    ': [-10, False], 'x': [100, True], 'obstacle':'#'}
```

4.2.2.2 State instance

As explained, the State class holds the actualized information of the grid and the position of the agent in that grid in each step. This parameter is the first State's instance from where to access to the Markov Decision Process state's network (environment). For instance, starting at position *(0,0)*:

```
start_state = State(grid=grid, agent_pos=(0,0))
```

It is remarkable that in environments with numerous states it is convenient to distribute the start state of the learning process, in this way the agent assimilates a more complete perspective of the problem and therefore, infers a more optimal path when facing any random practical case. For example, in the SEBA's evacuation case, the agent has been trained starting from every corner of each building's room.

4.2.2.3 'n_episodes' and 'n_episode_steps' parameters

In the call of the Qlearning's learning function it will be necessary to specify the number of episodes (stages of learning from the initial state) and the maximum number of possible steps per episode. The choice of the assigned values will depend on the environment dealing with, in particular, on the range of the different alternatives that the agent can potentially encounter during the exploration.
In order to value a first estimation it is convenient to compute the product of the number of Markov states and the number of maximum possible actions per state, as it gives the total amount of agent's alternatives to explore in that environment. See expression 4.1 to find this numbers on the SEBA's example presented in this section:

$$N^{o}\ actions\ \cdot\ N^{o}\ states\ =\ N^{o}\ alternatives \Rightarrow\ \mathbf{8}\ \cdot\ (\mathbf{20}\cdot\mathbf{20})\ =\ \mathbf{3.200} \tag{4.1}$$

Thereafter, to locate this parameters in the best optimality range, it is very important to correlate the total number of episodes with the total amount of alternatives. This implies a parameter tunning through a sensitivity study.
Figure 4.5 illustrates this concept with a minimalist view of this study in order to focus on the relevant facts. The vertical axis measures the optimality of the inferred solution, while the horizontal axis specifies the number of episodes runned along the learning process. The optimality is measured by computing the total reward resulted of an inferring process (all exploitation actions) at the end of each episode.
In the figure's legend, it can be appreciated the colours related to different number of states. Note that it could also have been changed the number of actions, the relevant fact is that the total amount of alternatives on the environment varies.
The coloured circles mark the number of episodes where that case reaches the optimal solution. On the contrary, the cross means that it failed inferring the optimal solution, although it possibly could have reached it by increasing the number of episodes.

For procedure simplicity, a hypothesis of the number of steps per episode is first selected. This value must surpass the number of steps an agent should carry out to reach the longest possible solution. For instance, in this section's example, it would be the longest route that communicates two points of the building floor, that is approximately 100 steps.

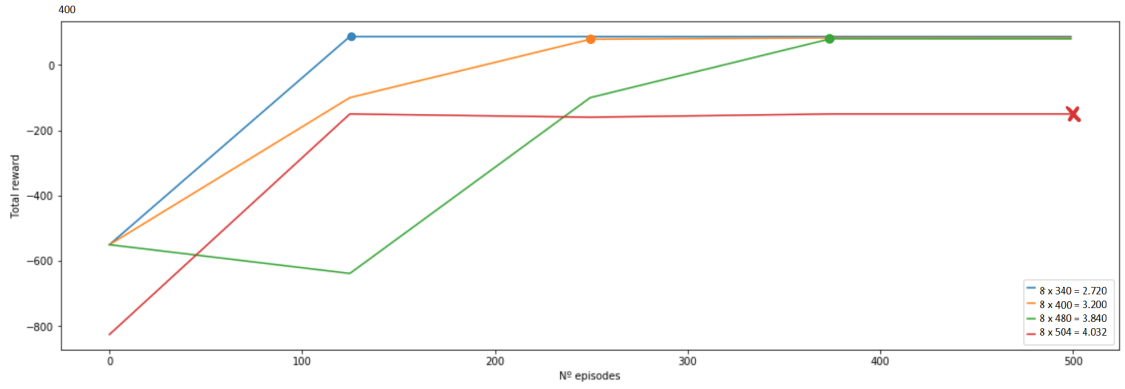

Figure 4.5: Number of episodes, optimality and number of alternatives correlation.

From the chronological points sequence it can be deduced that the more alternatives the environment has, the more episodes (and steps) needed to reach the optimal solution. The cross case could either reach an optimal solution if the learning process adds more episodes

or being limited by time and hardware's computing capacity. For the case of the section's example (8 actions and 400 states), an appropriate number of episodes is 300.

To conclude; on the one hand, a deficient number of episodes (***n_episodes***) will result in unexplored actions or a quality value with a weak and erratic representation of their suitability. On the other hand, an excessive number of episodes will suppose a waste of computing resources. Thereby, it is always preferable to choose this value upwards in order to minimize failing risks and then adjust it in the sensitivity study.

Now, with the same number of states and actions, the figure 4.6 illustrates the behaviour of the optimality depending on the number of steps per episode. As expected, values much lower than the longest possible solution hardly or never converge (see blue cross). On the contrary, as it increases the value, there comes a point at which the speed of convergence remains practically unchanged, however, the consumption of time and computing capacity continues increasing.
Going back to the example and considering the selected number of episodes 300, an appropriate selection range for the number of steps oscillates between 100-200, the choice depends on time and computing preferences.

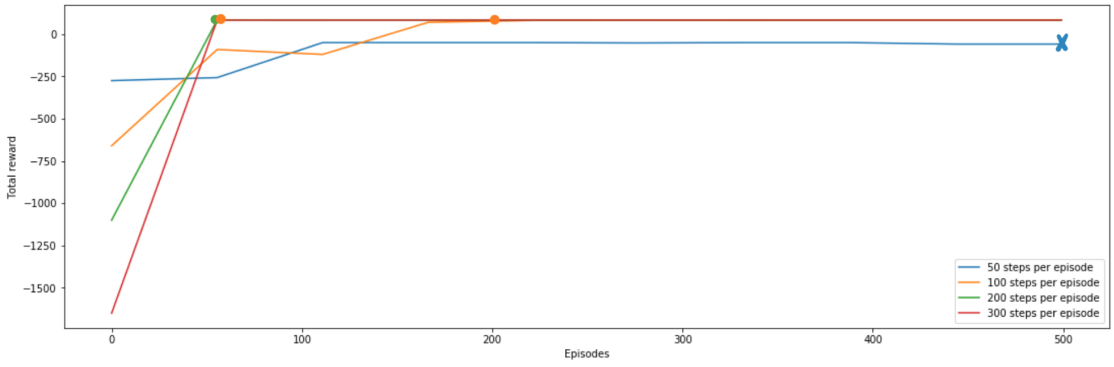

Figure 4.6: Number of episodes, optimality and number of alternatives correlation.

To sum up, for the number of steps (***n_episode_steps***), it is necessary to look for a value high enough to reach the number of states required to complete the optimal solution to be achieved, but not excessively high since it wastes computing resources while risking losing the perspective from which the agent faces each episode. That is to say, it will be better to concentrate the agent's experience around the first steps and not let him wander indefinitely transiting between the states, while conditioning the quality values to remote and irrelevant situations for the resolution of the problem.

4.2.2.4 Multi-agent approach

It is a common practice to vary the starting position from where the inference takes places. This method, precisely applies the first steps approach mentioned in the last paragraph of the previous subsection. As said, it consists on reinforcing the agent experience by being trained from several environment's perspectives. In such a way that each starting state must be chosen strategically to focus the learning process into the most relevant potentially simulated situations. The multi-agent name comes from the concept of a unique knowledge shared by multiple agents training processes.

In this case, the strategy is to equally distribute exploring views, for example by starting from every corner of the floor. The 'visualize_max_quality_action' created function perfectly fits in the demonstration of the mentioned process. It is a heat map representation of the grid, where the black cells can be labelled as unexplored, and the red range tonality coloured cells, as explored. Moreover, within the coloured ones, the lighter it is, the more suitable that cell is to achieve an optimal evacuation path. The aim of this function is to ensure the validity of the training strategy.

Figures 4.7 and 4.8 complement each other. The first one represents each starting point line convergence. The second one is composed of the 6 applied starting states heat maps. The intention is to visualize how the agent's knowledge spreads until it fills the whole floor. In the legend box on the down right corner of the plot, it is specified the starting state associated to each agent, as well as the confirmation that the agent who started the training from that certain state, reached the exit successfully.

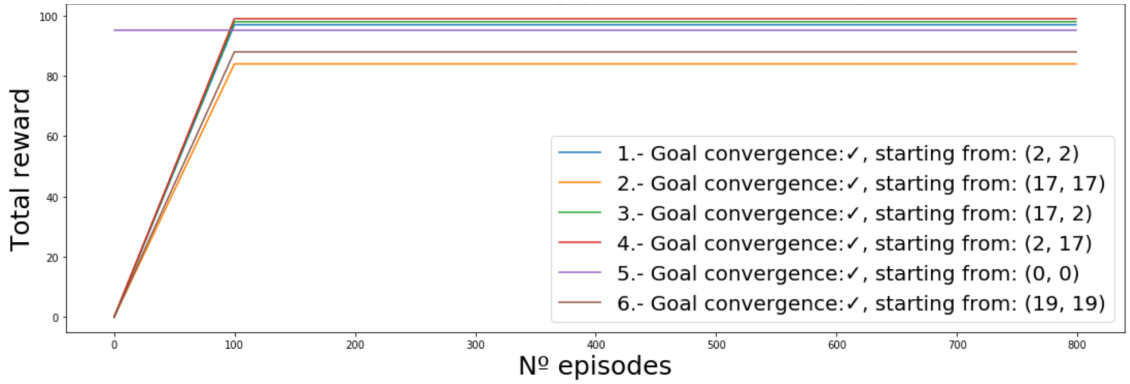

Figure 4.7: Multi-agent approach convergence.

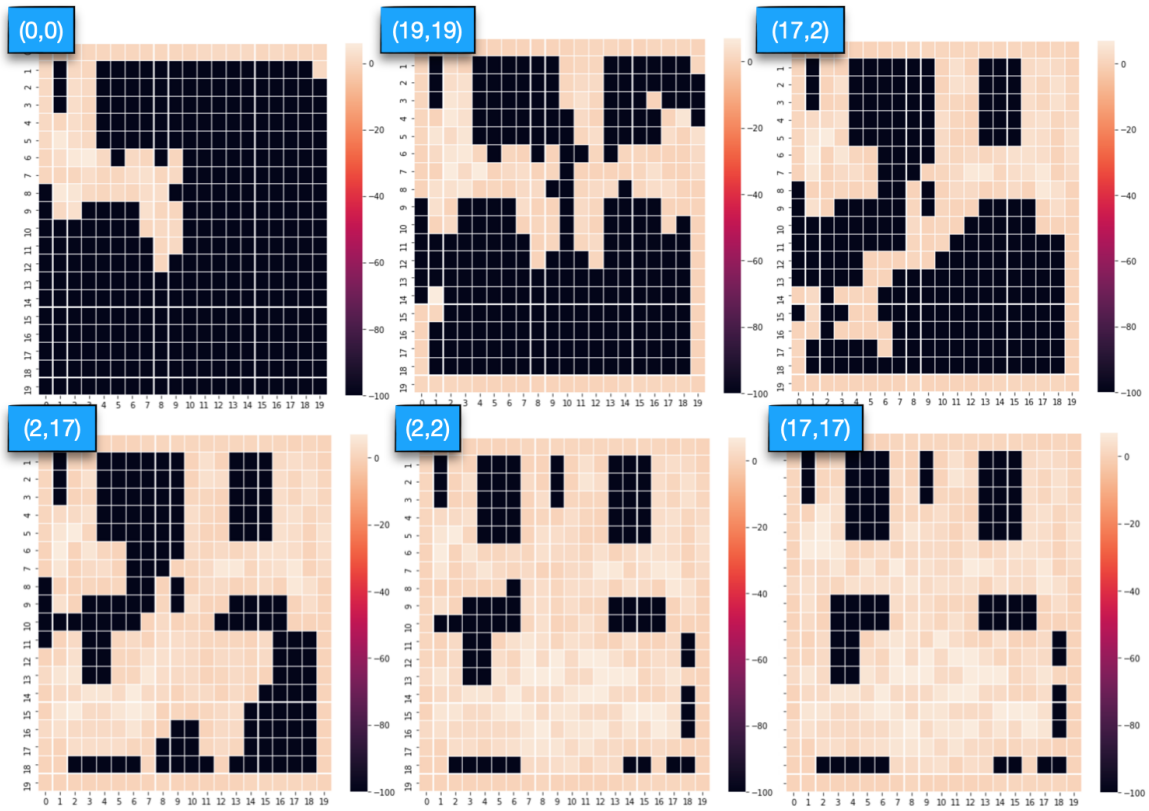

Figure 4.8: Multi-agent Q(s,a) sequential expansion along the grid.

4.2.2.5 QLearning instance

This subsection presents an overview of this instance's role in the Model class. Briefly described, it gives access to the learning and inference process through the Bellman's equation. Afterwards, in the QLearning class subsection, it can be found an in-depth study of itself.

To begin with, in the 'update_qlearning_files' attribute it must be specified if learning is needed (e.g. in case a new SEBA's building map has been uploaded) as a *boolean*. If true, the sequential diagram shown in figure 4.9 begins, triggered by the execution of the following code lines (note that all this parameters has been thoroughly explained in this section):

```
 e_greedy_maze = Qlearning(
start_state = State(grid=grid, agent_pos=(x,y),
grid_resources = grid_resources)

e_greedy_maze.learn(
n_episodes=n_episodes,
```

```
n_episode_steps=n_episode_steps)
```

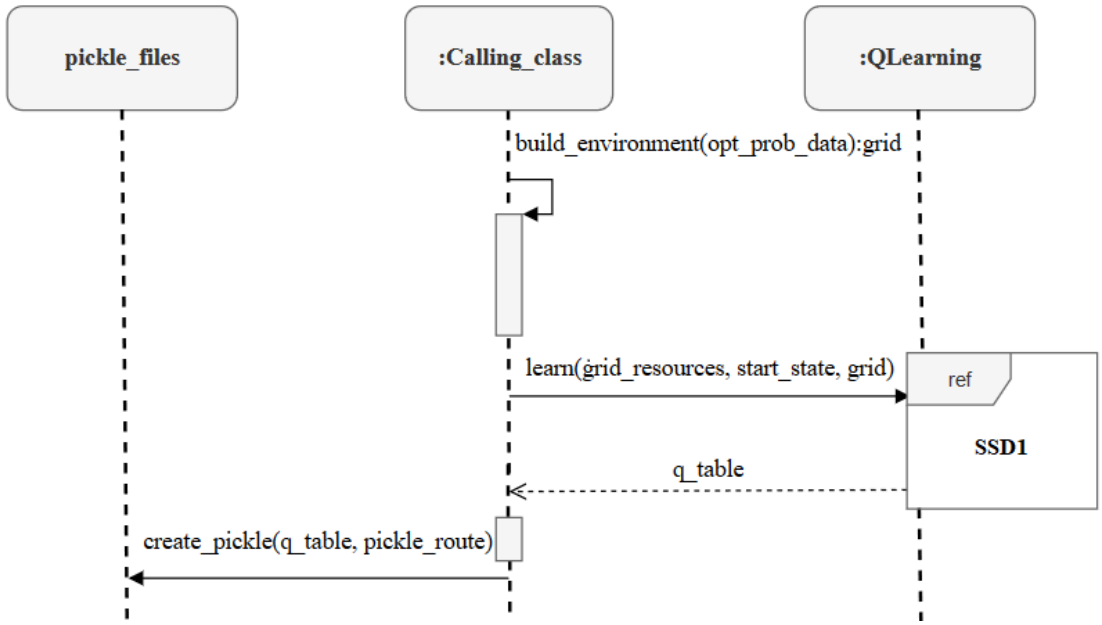

Figure 4.9: SSD of the overall learning procedure in Model class.

Figure 4.9 shows a simplified sequence diagram of the QLearning's learn function. The 'SSD1' reference points to next subsection's figure 4.11, which provides a much more detailed sequential diagram explanation of the learning procedure taking place inside the QLearning class.

Afterwards, along the simulation, the inferring function can be instantly called by the execution of the following lines, returning the most optimal solution instantly (less than 3 ms):

```
 e_greedy_maze = Qlearning(
inference_state = State(grid=grid, agent_pos=occupant.pos)
grid_resources = grid_resources )
path, tot_reward = e_greedy_maze.infer_path(n_episode_steps,
    inference_state)
```

As well, at figure 4.10 the 'SSD2' reference points to the next subsection, but to the inferring procedure 4.13 instead.

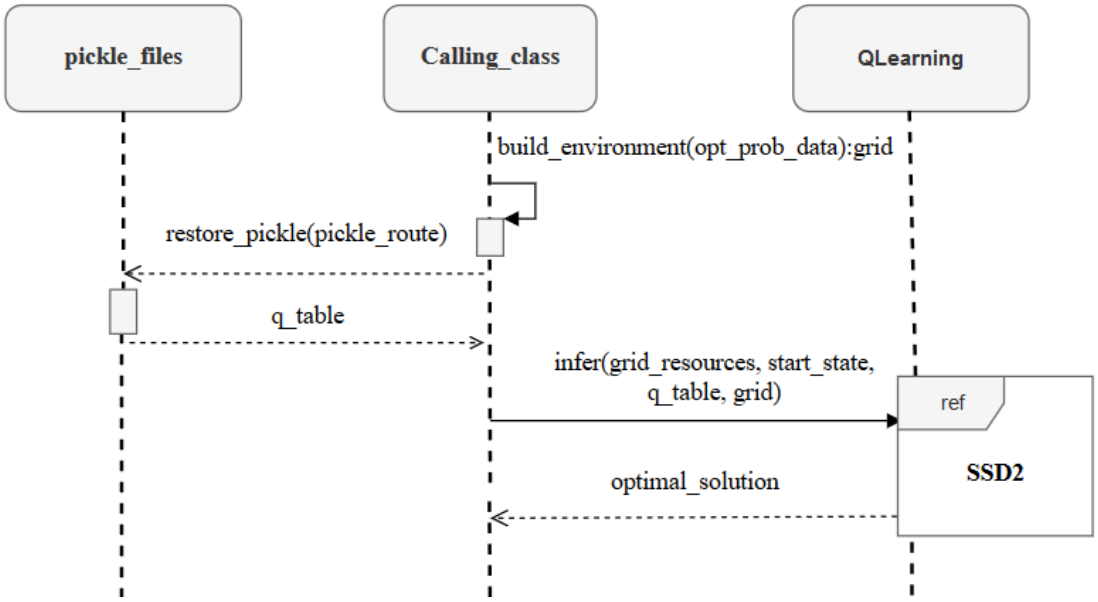

Figure 4.10: SSD of the overall inferring procedure in Model class.

4.2.3 QLearning class

Finally, let's dive into the mechanism behind these procedures: the Bellman's equation. To review this concept go back to equation 2.9. The squared code detailed below represents the used way to program it in this project. All the functions declared in this class spin around this equation.

```
self.q_table[state][action] = self.q_table[state][action] +
alpha * (reward + self.gamma *  np.max(self.q_table[next_state]) - self.
    q_table[state][action])
```

The description of this class must include each of its functions as it plays the most relevant role of this project. Moreover, this explanations will provide a better diagram's pseudo code comprehension. Starting with the main ones:

- ***observe_reward_value***: receives the grid's symbol stepped on and gives its respective reward. Its *boolean*'s value determinates the possible ending of the current episode.
- ***extract_possible_actions***: actions along the grid are defined here. In this case as spatial movements directions. If necessary they can be re-designed for other applications.
- ***choose_action***: this applies the E-greedy policy strategy when selecting the action (exploration vs exploitation). It could be possible to be changed for further utilities,

for example, to Boltzmann's strategy.

- ***learn***: Bellman's equation process. This function coordinates most of the previously described ones and fills the *q_table* attribute.

- ***infer_path***: deduction of the optimal policy from the *q_table* with a given initial state and maximum range of steps.

To continue with, the list detailed below includes all the secondary functions used for sensibility analysis and visualization purposes:

- ***visualize_inferred_path***: *string* logged representation of the states resulted as optimal policy.

- ***visualize_max_quality_action***: Seaborn and Matplotlib coloured representation of the maximum quality value in each state.

- ***visualize_n_episodes_sensibility***: Matplotlib representation of the correlation between the optimality convergence and the number of episode.

- ***visualize_steps_sensibility***: Matplotlib representation of the correlation between the optimality convergence and the number of steps.

- ***ASCII_q_values***: translates the resulted quality values into Unicode characters.

- ***warn_if_obstacle***: throws exception if the given starting position of the agent was chosen over an obstacle.

- ***convert_to_pickle*** and ***extract_from_pickle***: this two methods are used to enable persistence of the 'q-table' dictionary by converting/extracting it in pickle form.

Once explained, let's see how all these functions are orchestrated inside the learning and inferring ones , respectively shown at the figure 4.11 and 4.13.
The blue boxes are an abstraction to make it easier to understand. In the real code, the agent is represented by the QLearning class where the algorithm is defined and the environment represents all its information, which at least, it must include the ***grid*** and ***grid_resources*** parameters.
To review the *epsilon*, *gamma* and *alpha* concepts go back to subsections 2.4.1.1, 2.4 and 2.3 respectively.

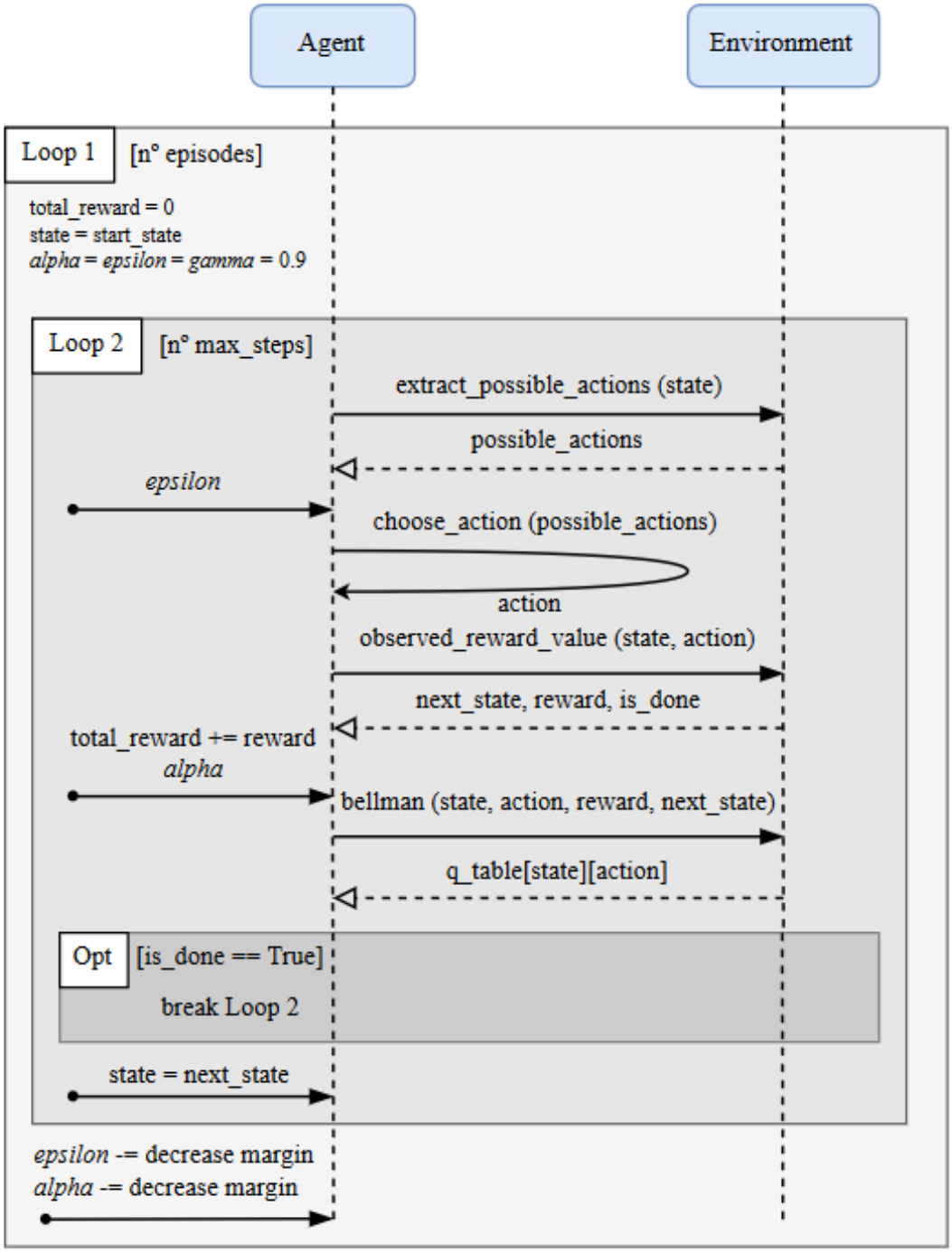

Figure 4.11: SSD1: in-depth learning procedure of QLearning class.

A partial sample of the *q_table* resulted from this process is shown below. In order to verify its coherence see figure 4.12. This rendering is printed from the ***'ASCII_q_values'*** function, on QLearning class. Brief description of the used Unicode characters; the arrow represents the maximum quality value of each state, the black squares are obstacles and

the black circles are exits. It can be clearly recognised the flows of the directions heading towards the exits while avoiding the fire and obstacles. Afterwards, in the inferring sequence diagram explanation, the yield of this resulted variable will be tested on several random real practical examples on SEBA's simulation.

```
{<e_greedy_q_learning.State object at 0x1a0c367c88>: array([79.99   89.
    89.99   88.   87.99   90.  88.  89.99]),
<e_greedy_q_learning.State object at 0x1a0c367710>: array([91.   82.    92.
        90.    89.99   92.   90.   83.  ]),
<e_greedy_q_learning.State object at 0x1a0c367630>: array([88.   87.99  89.
        87.    86.90   79.99   86.99  -0.98]),
...}
```

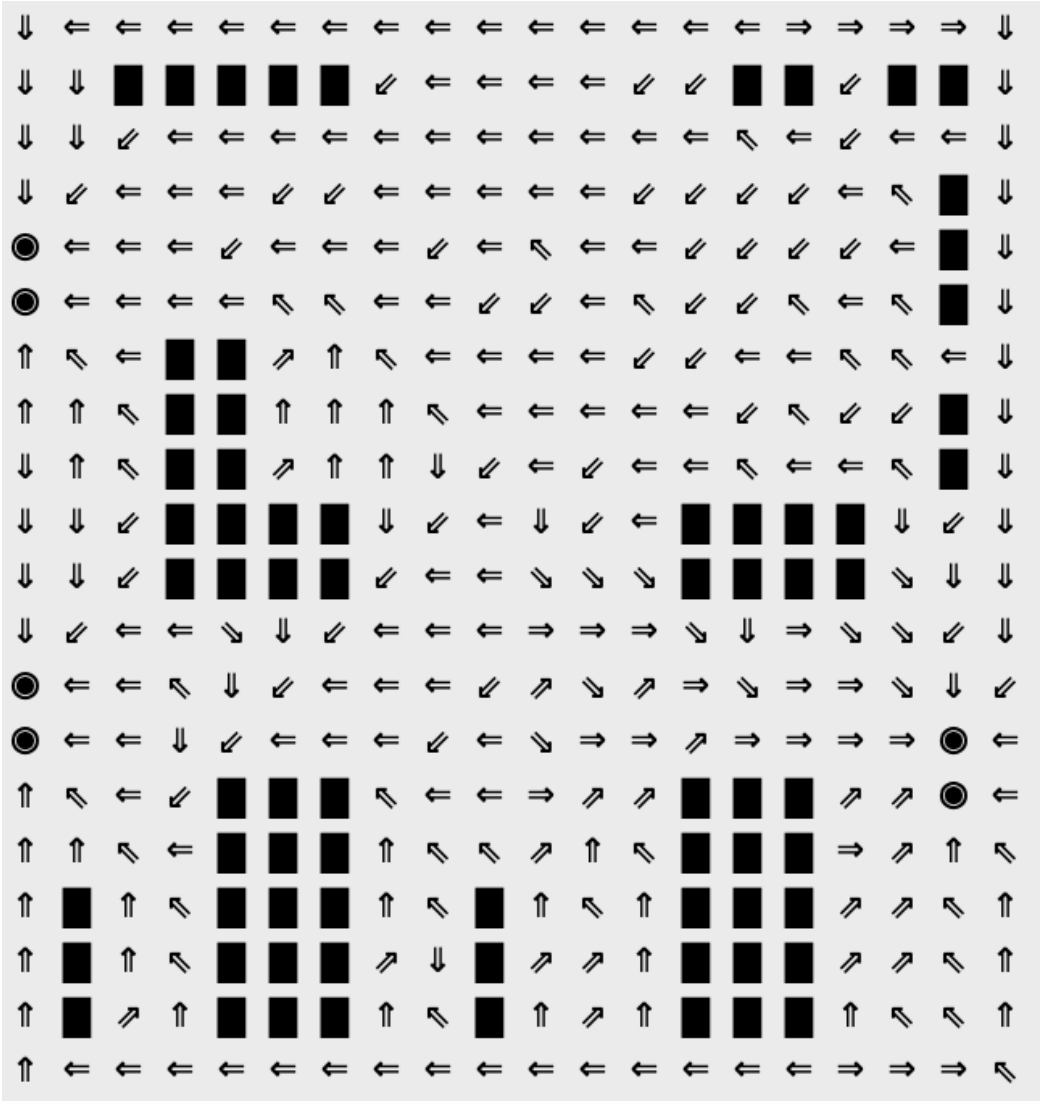

Figure 4.12: Unicode most optimal action per cell.

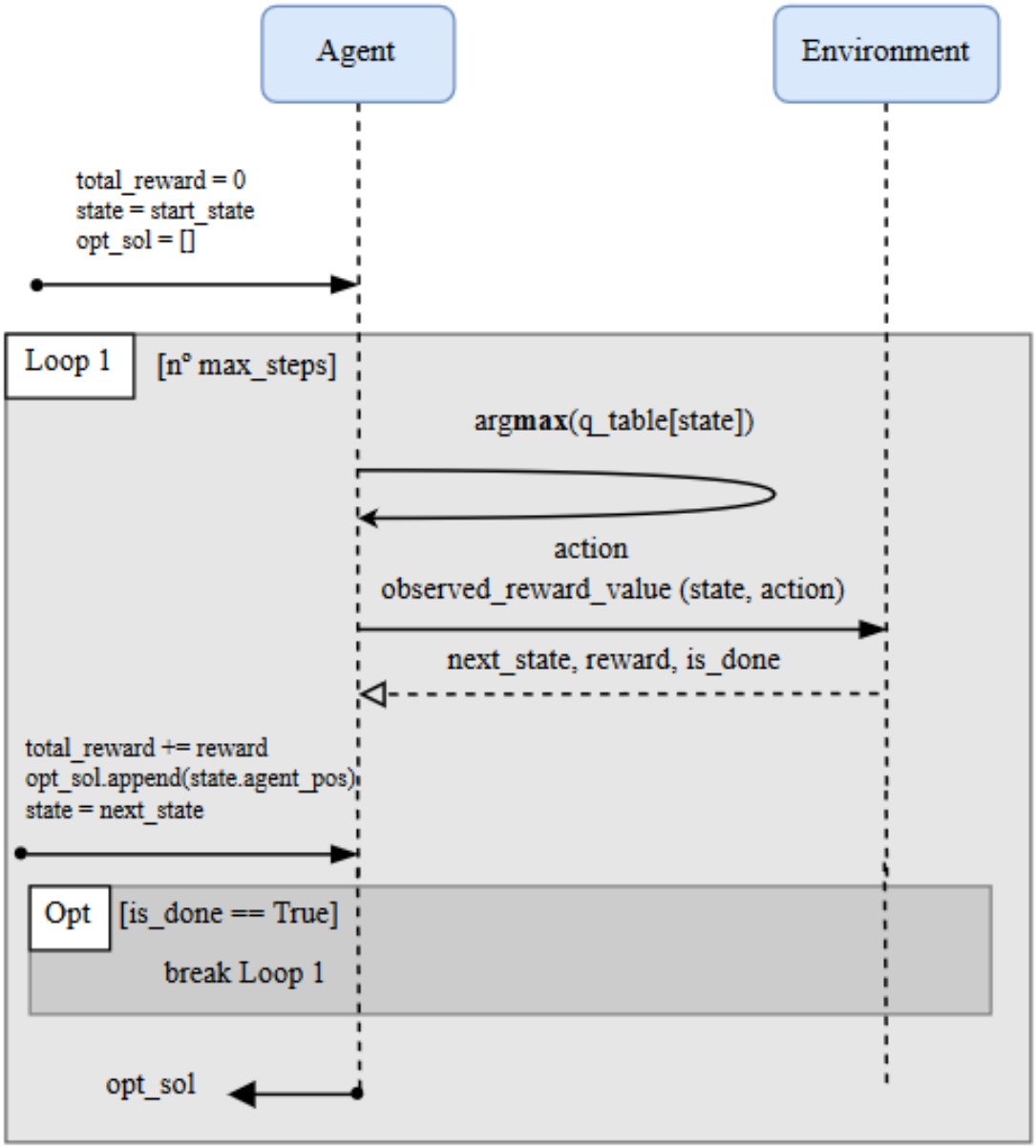

Figure 4.13: SSD2: in-depth inferring procedure of QLearning class.

In figure 4.14, it can be visualized the final result of this section's example, through the application of the inferring process to the obtained *q_table* to evacuate the agents in this specific building simulation. The agents where walking randomly along the grid when the fire distribution suddenly appeared. Afterwards, the scape route was instantly computed from the last cell they stepped on (obtaining a Q-learning optimal solution per agent). The complete followed path until reaching the door can be visualized with the blue discontinue arrows. It can be verified that the chosen way coincides with the flow showed at fig. 4.12.

Note that all the routes are the shortest adn safest among the possible thanks to the negative reward given by default to each coordinate in order to penalize distance and fire. As any agent was burned, the survival rate was 100%.

If verbose mode is on, in the console it appears the logged information seen at 4.15. Printing a string representation of the followed path per agent represented with '@' symbol (instead of blue arrows) shown in 4.14. The rest of the characters appearing in the 2D string grids had already been explained at 4.2.2.

Also, in the code's box located below these images, there can be appreciated the printed details of the final agent's state-action quality values per faced state, their action choice (the one with the maximum total reward) and the state it leaded to.

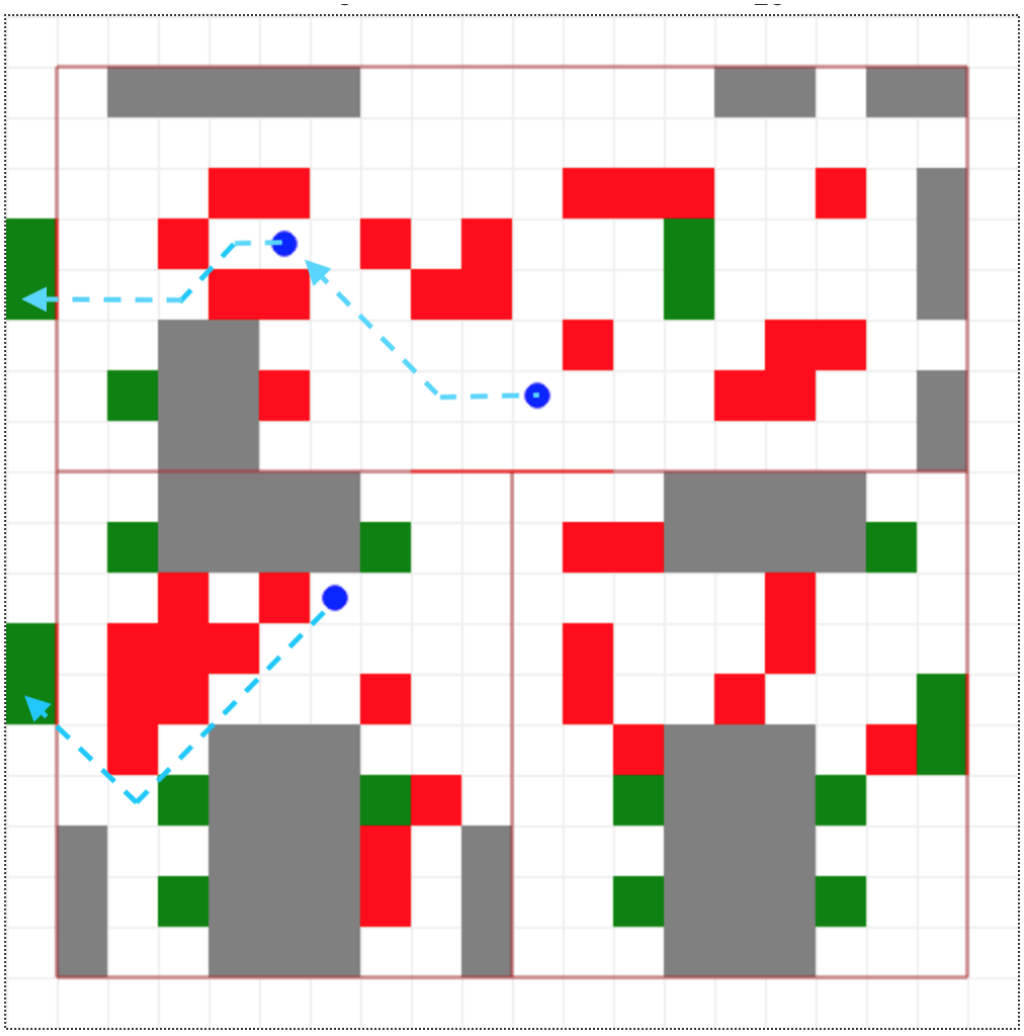

Figure 4.14: SEBA's visualization of three agent QLearning fire evacuation cases

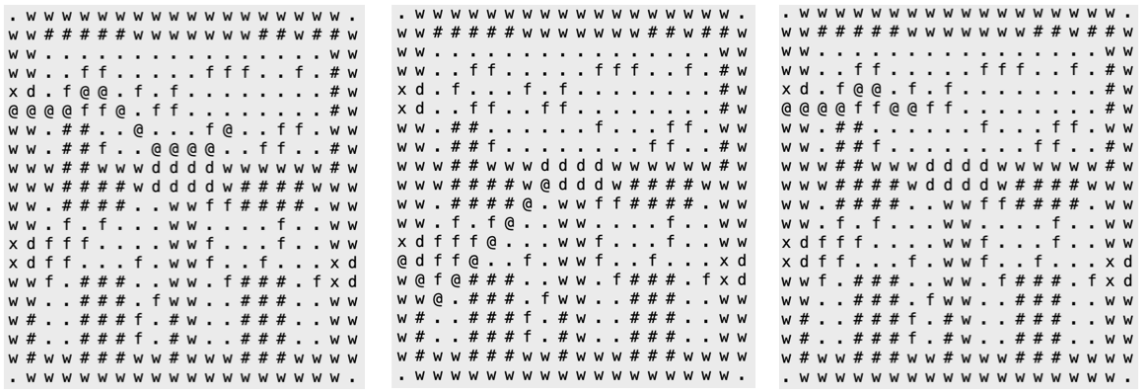

Figure 4.15: Verbose mode printed representation of the followed path.

```
Q(s,a)= [88. 88. 80. 86. 87. 89. 87. 89.] => a = 5, s = (12, 13)
Q(s,a)= [89. 80. 90. 88. 88. 90. 88. 90.] => a = 5, s = (11, 12)
Q(s,a)= [90. 90. 91. 89. 80. 91. 89. 91.] => a = 2, s = (10, 12)
Q(s,a)= [91. 91. 92. 90. 90. 91. 90. 92.] => a = 2, s = (9, 12)
Q(s,a)= [91. 92. 92. 91. 91. 91. 91. 93.] => a = 2, s = (8, 12)
Q(s,a)= [92. 93. 93. 92. 83. 92. 92. 94.] => a = 7, s = (7, 13)
Q(s,a)= [93. 94. 86. 93. 84. 93. 93. 95.] => a = 7, s = (6, 14)
Q(s,a)= [86. 86. 96. 94. 94. 87. 94. 87.] => a = 2, s = (5, 15)
Q(s,a)= [87. 87. 88. 95. 86. 97. 86. 97.] => a = 5, s = (4, 15)
Q(s,a)= [88. 98. 87. 96. 98. 98.] => a = 1, s = (3, 14)
Q(s,a)= [98. 98. 99. 97. 88. 99. 99.] => a = 2, s = (2, 14)
Q(s,a)= [99. 99. 100. 98. 98. 99. 98. 100.] => a = 2, s = (1, 14)
Q(s,a)= [98. 99. 97. 96. 98.] => a = 2, s = (0, 14) => exit reached
Evacuation route:[(12, 13), (11, 12), (10, 12), (9, 12), (8, 12), (7, 13),
    (6, 14), (5, 15), (4, 15), (3, 14), (2, 14), (1, 14), (0, 14)]
```

4.2.4 Occupant class

Whenever SEBA's emergency mode is activated due to fire recognition, if the evacuation strategy variable is 'q_learning', it provides the current position of each agent from where to compute the inference procedure. Once obtained, evacuates all agents through each respective route as explained before.

CHAPTER 5

Case study

5.1 Introduction

This chapter exposes an extension of the method presented in the previous chapter: evacuation of the building with static fire, where the agent dodged the fire as if it was an obstacle, since its location did not change either during or after the learning process. Now, the fire appears in random positions of the room, expanding as time passes.

5.2 Application of Q-learning to a dynamic fire evacuation

There are many ways to apply Q-learning to this situation. The following solution has turned out to be one of the most efficient ones. Additionally, it has also served to draw potentially valuable conclusions. In particular, despite its good results, it defines the limitations of Q-learning and opens the door to the introduction of Deep Q-learning. This technique will be addressed superficially in the Appendix C to establish solid foundations from which to develop more optimal contributions (specficied at Future Work sec. 6).

5.2.1 Viability analysis

The best way to understand the problems presented in this situation is to start by analysing the major limitations with which our agent (the algorithm) encounters in this new case.

Before using any algorithm it is necessary to approximate the expected computing resources: time and memory, in order to anticipate unviable results. This step will be used in the following paragraphs to present one of the main enemies of Q-learning: scalability.

To carry out the inferring successfully, the agent should register all the different possible combinations of cells with fire. For example, in the floor designed in the previous chapter, for each cell where the agent can be placed, there would be 399 remaining permutations to explore (order matters). Omitting the obstacles, this calculation results in an approximate magnitude of $\boldsymbol{P_n = n! = 400! = 10^{800}}$ different cases, where $\boldsymbol{n = 400}$ states or cells.

As if that value wasn't high enough, also, remember that for each of these situations the agent must execute an average of 300 episodes (value concluded in the previous chapter 4.6) to deduct the quality values. Thereby, the resulting needed learning time and memory capacity would be unreachable.

To solve this issue it is necessary to drastically reduce the number of combinations that has to be learned. That is, generalize in larger groups the possible situations in which the fire may appear in the room. In such a way that minimizes the variation of the scenario, while maintaining a coherence in the learning that enables a satisfactory inference.

For example, if we group the cells in the surroundings of each of the floor's exits, the reaction of the agent when he finds fire in any cell distribution near a door could be summarized to the same one with only that door burning. In most cases that door would be discarded and consequently the agent should recompute the optimal path. See figure 5.1 to visualize the explanation.

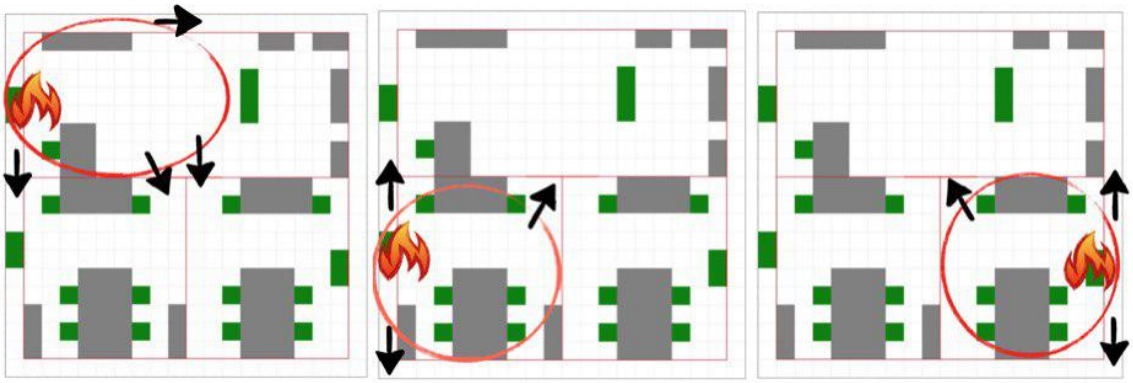

Figure 5.1: Example of the generalization of all possible fire combinations.

In this way, the number of combinations of the scenario that the agent must learn is simplified to the combinations generated among the number of doors the room has. Thereby, as the example's floor has 3 doors the agent should store in its memory 7 maps with these enabled doors respectively: 1, 2, 3, 23, 13, 12, 123. So each time the fire appears in front of the agent, it will be considered that the door to which it was directed is disabled, changing its quality value map 'in mind' to the one which substitutes that door with fire.

5.2.2 Logistic implementation

The occupant class is the orchestrator of this process. On the one hand, it handles all the occupant's position and movement along the floor, supervising the appearance of fire in their surroundings. On the other hand, it calls the Q-learning class inferring function whenever the evacuation demands it. If fire irrupts in the inferred evacuation path of any agent, the final exit associated to that path is discarded for all the agents in the floor. This is accomplished by updating the grid used to access the respective action's quality values. Figure 5.2 illustrates the explained concept.

Additionally, the whole process is summarized in the activity diagram shown in figure 5.3. Note that the main function of all three agent simulation frameworks is called *step*, and works as an inner clock within the simulation. This function is called at equally distributed instants of the time line, and within that time interval the simulation carries out all the environment and agent's performance. For example, it triggers the stochastic Markov transitions that rules each agent activity.

To easily understand this procedure let's compare the reasoning of a real person in a building evacuation context with the SEBA's implementation specified between brackets. Fire appears at an unknown location of a floor and the fire sensor detects it (SEBA creates an array with the current coordinates of fire per step). The alarm of the building is activated in order to warn the occupant (SEBA's emergency state is changed to *true*). The occupant evacuates the building through the shortest route (Q-learning inference takes place and stores that path in each agent's instance). However, in the run, the agent suddenly encounters fire (the *verify_path* function warns if the inferred path intersects the array of fire). Consequently, he discards the headed exit and thinks about the next closest one (the used 'q-table' is changed to the one that substitutes that door with fire).

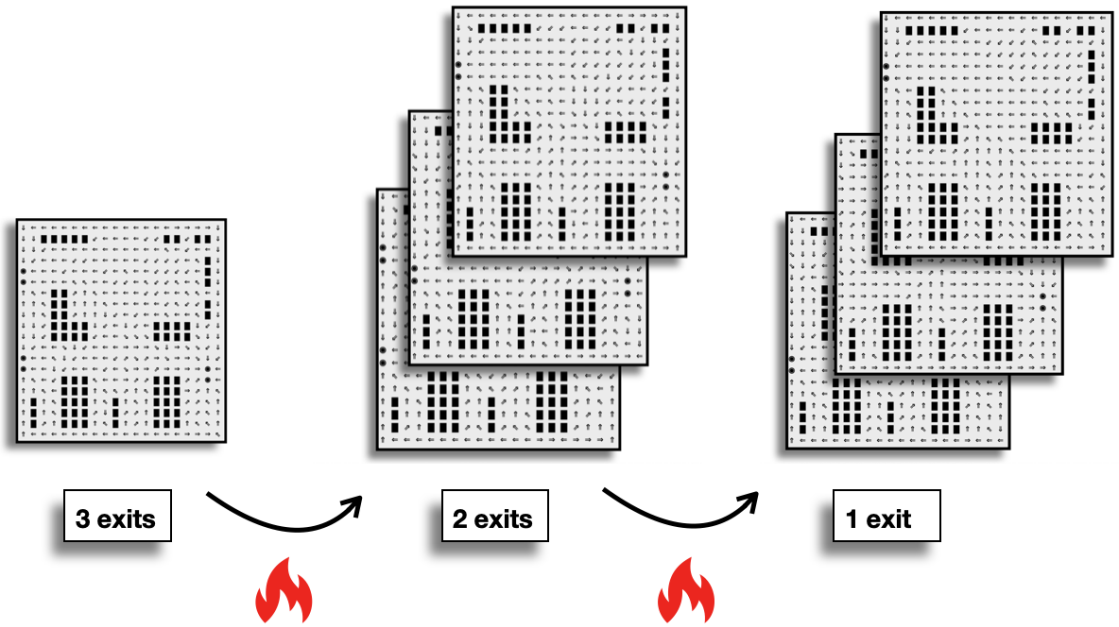

Figure 5.2: Agent grid migration process.

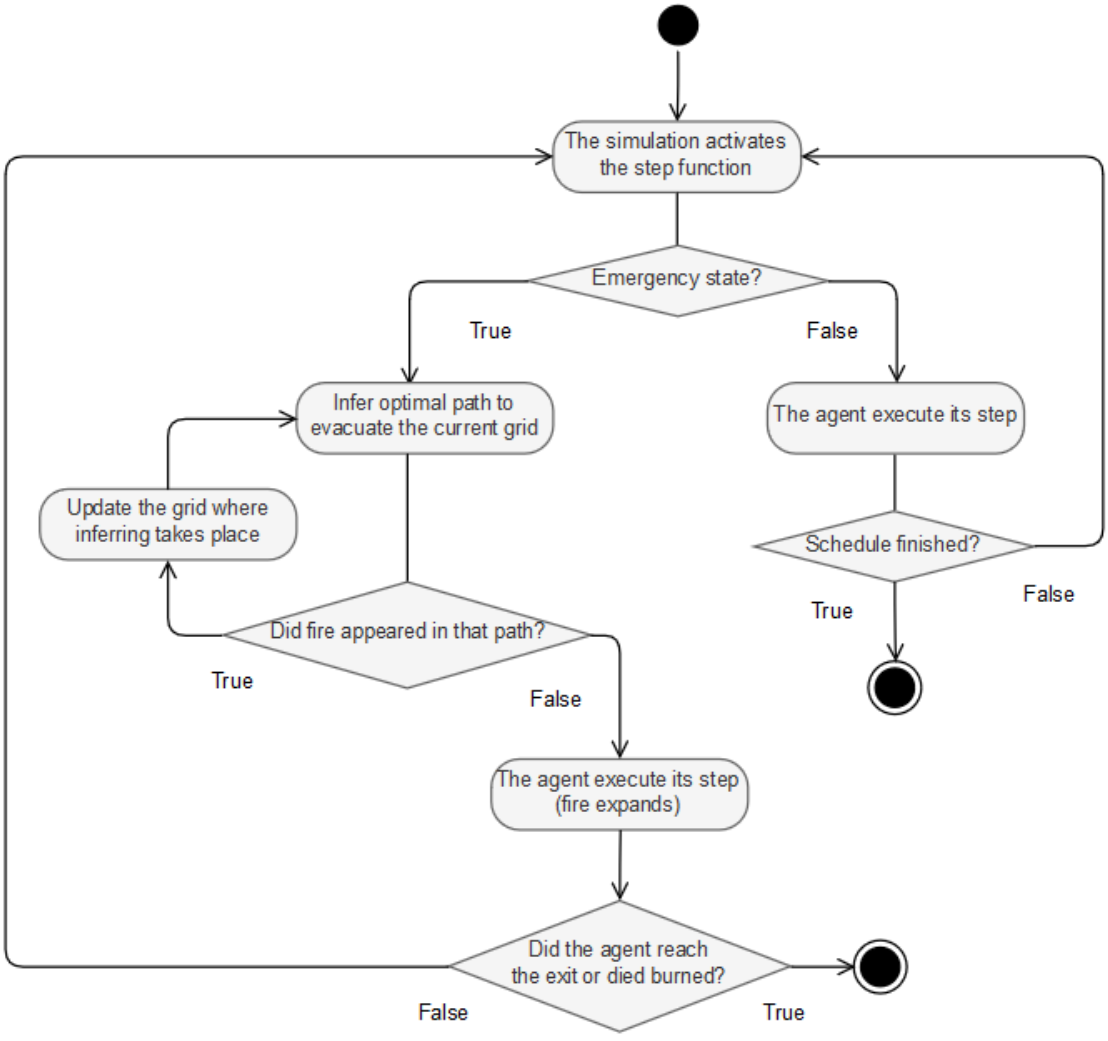

Figure 5.3: Activity diagram of the implemented evacuation procedure.

5.2.3 Simulation results

This subsection shows the results of two sample simulations: 5.4 and 5.5. To begin with, have a look to the blue squared grid, it defines the followed final path to scape from fire by each of the agents. The green squared grids are chronologically ordered, from upper left to down right. Alternatively, the sequence can be followed by considering the fire expansion. Note that the fire appeared randomly in any cell of the floor, and the agents applied the explained method to decide the safest and shortest route. In every case the survival rate has been 100%. For further detailed statistics read next subsection sensibility analysis: 5.3.

In case 5.4 it has not been necessary to recompute any path, because the first choice reached the exit without encountering the fire for all five agents. In contrary, at the 5.5 case, the agent pointed by the wider blue arrow, is forced to recompute the escaping path in order to avoid the fire.

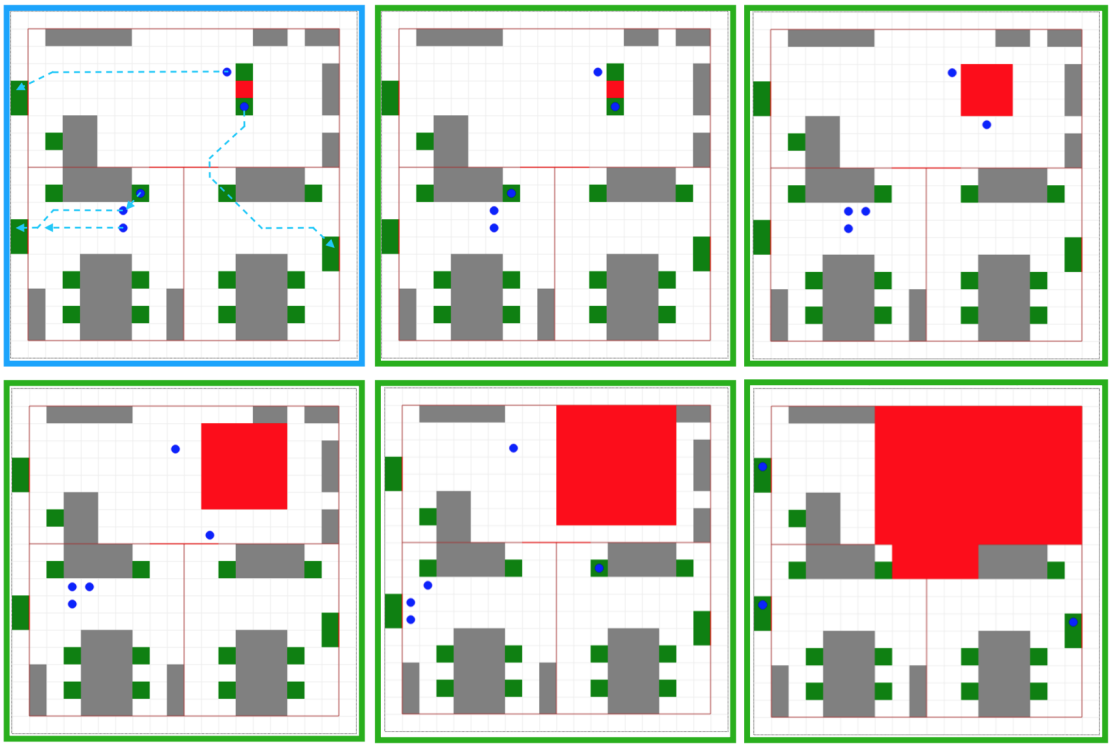

Figure 5.4: Simulation 1 of a dynamic fire evacuation with Q Learning.

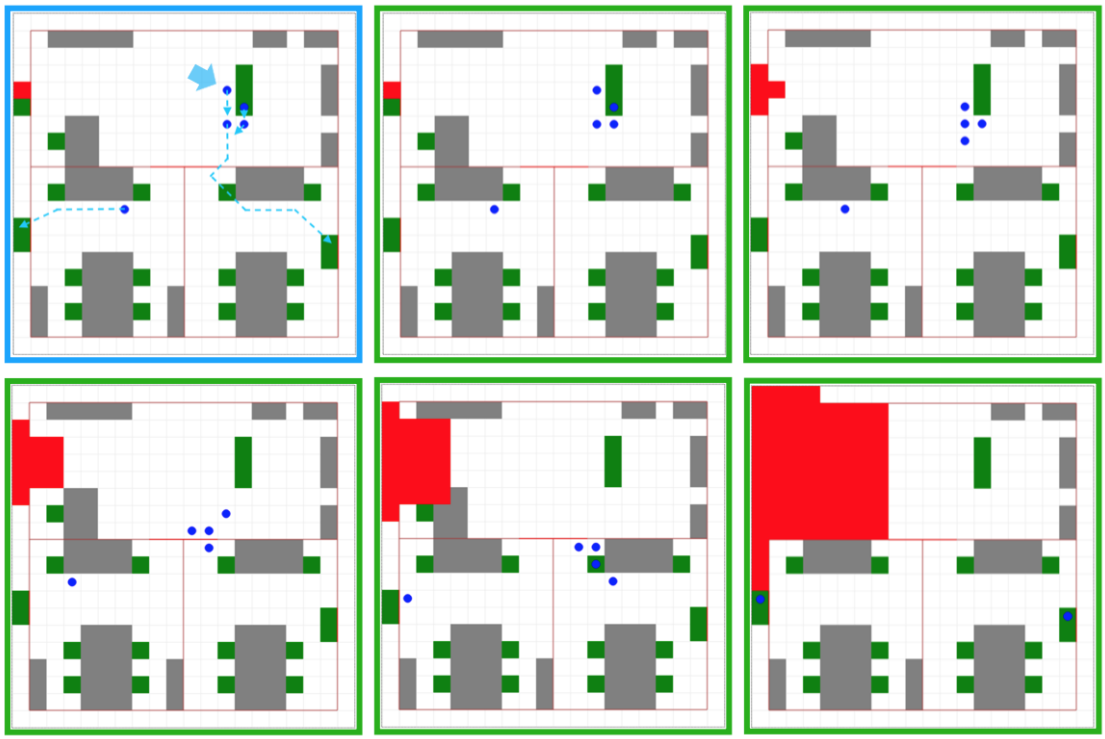

Figure 5.5: Simulation 2 of a dynamic fire evacuation with Q Learning.

5.3 Sensibility analysis

5.3.1 Comparison among evacuation strategies in SEBA

SEBA already includes the implementation of three strategies to scape from the fire. The difference between them lies on the applied criteria when choosing the correct exit, which are:

- ***'Nearest'***: Choose the nearest exit from the agent position in the building floor, regardless of the fire distribution.
- ***'Safest'***: Choose the exit whose path is the farthest away from fire.
- ***'Uncrowded'***: Choose the exit whose path has fewer people, regardless of the fire distribution.

All of them uses the A star algorithm to compute the optimal mathematical distance from the current position to the selected exit according to the strategy. Therefore, in order to obtain an accurate analysis of the impact that Q-learning evacuation strategy has on SEBA's simulation, it is necessary to carry out a thorough comparison between this two path-finding methods.

5.3.1.1 A star versus Q-Learning

To begin with, let's see how the **computing complexity** of these algorithms respond to the growth of the number of states on the building floor.

- The **Q-learning algorithm** complexity can be dived into two parts: on the one side the training, which takes place before the application to the optimization problem and on the other side, the inference synchronized with its resolution. In this way, the computational complexity of the first path does not matter at all as it does not affect directly to the simulation process. Following the Big-O notation it could be approximated with a quadratic performance $\boldsymbol{O(n^2)}$, where **n** denotes the number of states necessary to reach the targeted cell (the shortest path). On the other hand, the inference part reduces this cost to a linear $\boldsymbol{O(n)}$ complexity. This last part is synchronized with the resolution of the optimization problem, thereby it will be the one used to represent the Q-learning algorithm computing complexity.

- The **A star algorithm** splits recursively in tree shape until reaching the targeted cell. Therefore, the number of nodes expanded is exponential in the depth of the solution (the shortest path). Being represented by $\boldsymbol{O(2^n)}$, where **n** denotes, as well, the number of states.

The three mentioned shapes can be observed on graph 5.6. The computational efficiency of the Q-learning algorithm is remarkable higher. In other words, the amount of memory (RAM) needed, as well as CPU capacity and time involved in the optimal resolution is considerably fewer.

The total computing time for the implemented Q-learning algorithm can be approximated to 0.03 seconds in an ordinary CPU. The A star algorithm oscillates over 0.6 and 0.7 seconds. The overhead of SEBA's framework suppose an extra time of 0.1 seconds. The computing time comparison among all the four strategies collected with SEBA's samples is illustrated at graph 5.6.

In order to obtain a complete analysis of the suitability of each algorithm, let's now see which one gives the highest survival rate. Figure 5.8 has been built for this purpose, the vertical axis measures the sum of the averages of deaths among 20 simulation trials for 18, 13, 7 and 3 agents. There are four columns, one per evacuation strategy composed of four colours, according the number of agents with which the simulation was carried out. Each coloured brick measures the contribution of a set of those 20 trials with a certain number of persons over the average deaths. To continue with, figure 5.9 has been generated from the average survival rate of the four number of person sets per strategy.

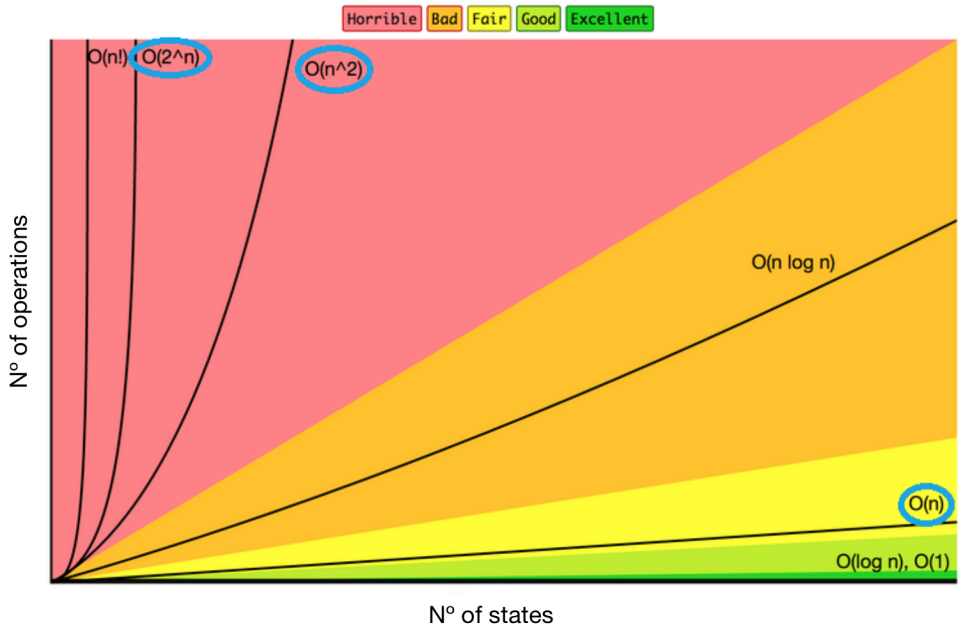

Figure 5.6: Big-O notation computing complexity

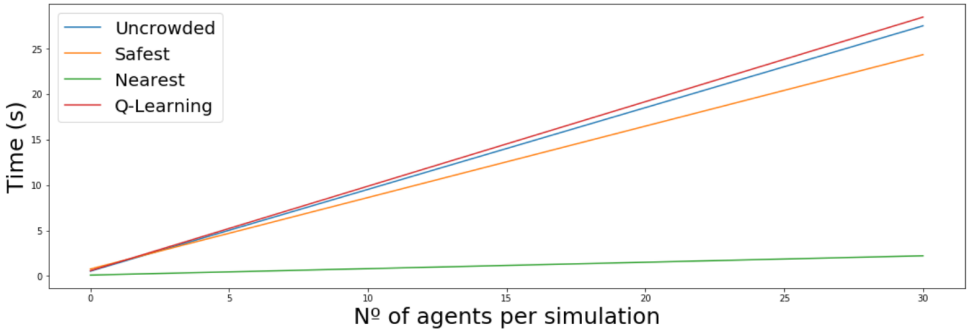

Figure 5.7: Computing time per strategy scaled by the number of agents

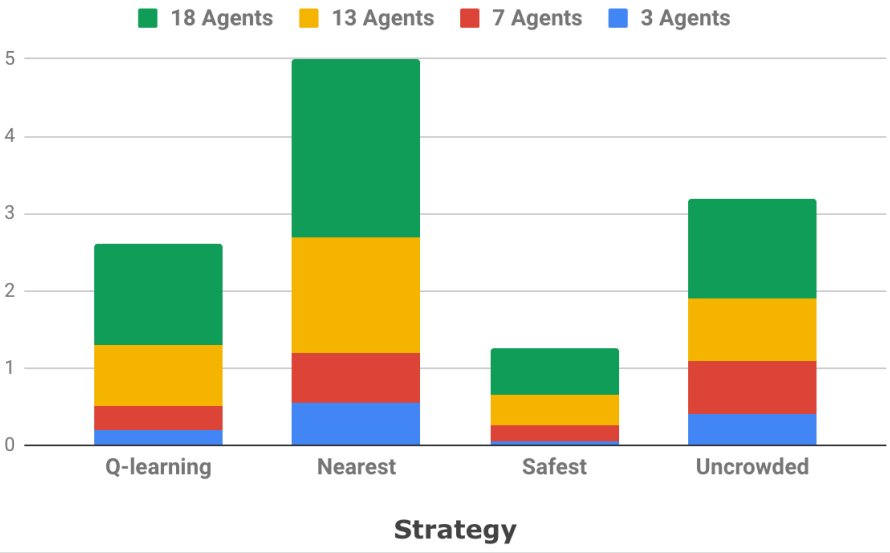

Figure 5.8: Average deaths per simulation with a variable number of agents

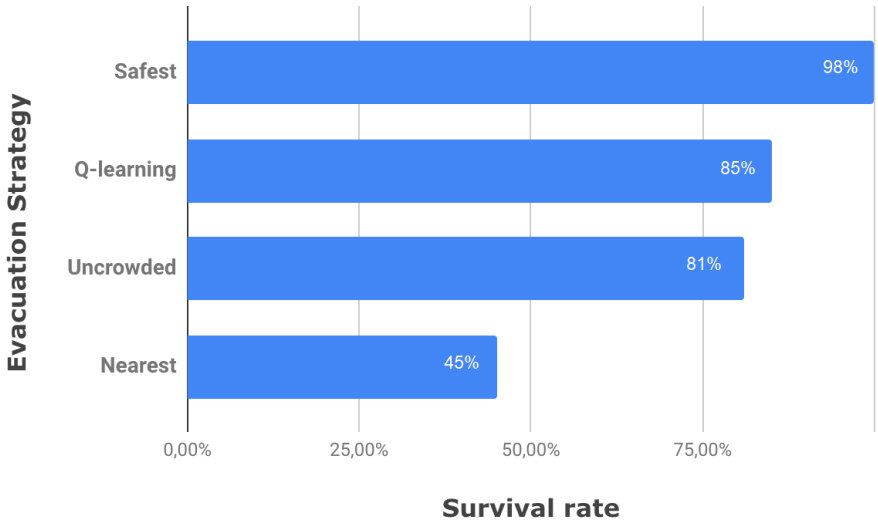

Figure 5.9: Survival rate as alive persons from total number of persons.

The Nearest strategy lacks consistency since it goes to the nearest exit without considering the existence of fire, consequently almost the half of agents step into fire.
The Uncrowded strategy improves as the number of people evacuated increases, although it offers better results, nor does it consider the location of the fire when inferring the escape route.
Certainly, the Safest strategy offers the highest rate of survivors. But this is because there is only a single focus of fire, and it evacuates the people through the farthest path, from the moment it appears on the floor. Therefore, it is practically impossible to run into it again. A second source of fire would ruin its evacuating logic if the escaping 'safest' path from one fire is irrupted with the other fire.
The Q-learning strategy provides the greatest flexibility of environment with a very high survival rate for any context with any number of fire simultaneous origins thanks to its dynamic logic which adapts according the context changes. Also, it has a lot of potential to scale, incorporating additional considerations, like choosing the less crowded exit among the other criteria. For more complex implementations such us social profile or psychological facts it would be necessary to introduce a Deep Neural Network (see sec. 6.3 and Appendix C).

CHAPTER 6

Conclusions and future work

This chapter summarizes the results and learnings of this project and proposes some points to develop as future work.

6.1 Conclusions

The results obtained in this project have demonstrated a significant improvement at SEBA's functionality, providing an much more efficient lightweight evacuation alternative to the A star choice. In addition, the versatile and generic way in which the Q-learning tool has been implemented will allow an easy adaptation to additional applications. Anyhow, the value of this project goes further with the future work approaches, heading the research towards new lines of study that can enhance the applicability and usefulness of the SOBA and SEBA frameworks to an unprecedented extent.

6.2 Achieved goals

- Implementation of a Q-learning module in an open format to reuse it at multiple applications. Its performance is better than most of the external offered libraries (evaluating computing time, flexibility and results).

- Clean integration of this module into SOBA and SEBA frameworks.

- Achievement of the simulation evacuation process through Q-learning algorithm.
- Reduction of the evacuation computing time up to 10 times comparing with the others SEBA's strategies.
- Successful joint of two of the other three strategies at once: safest path and nearest exit.
- Proposal of potential applications to integrate more powerful Deep Learning techniques.

6.3 Future work

The following list exposes the main proposed study lines to extend this project. The supporting argumentation behind these ideas can be found at the Appendix sec. C.

- Integration of Deep Neural Network techniques into Q-learning algorithm, such as CNN, to reduce the amount of memory and time needed to train an agent.
- Experimentation with Deep Q-learning libraries (Tensorflow, Keras, etc.) by adding SEBA the functionality to perform its simulation as a game format framework.

These two approaches could drastically propel the computational efficiency enabling much more complex applications, such as:

- Incorporation of a much realistic and richer social network simulation (SEBA).
- Implementation of new energy efficiency techniques (SOBA).
- Use of more optimal network connectivity methods (SOBA).

Additionally, it may be interesting to join forces with new technical areas, such as architecture. This fact could reinforce and provide new perspectives from where develop several promising study lines.

APPENDIX A

Impact of this project

This appendix exposes economical and social (current or future) impact that the work carried out in this thesis may suppose.

A.1 Introduction

Artificial intelligence can easily complement building security systems, saving many costs, not only human, but also economical. Its application can be of two types; prior to the occurrence of the emergency event or while it is occurring. On the one hand, in the first situation, it provides a simulation of platforms that recreates the circumstance from which the vulnerabilities are intended to be extracted. On the other hand, enables the management of the emergency in real time, in order to help solve complex situations with multiple alternatives instantly.

Specifically, this project focuses into applying a correct execution of the evacuation within constructions in order to avoid multiple potential tragedies: terrorist attacks, fires, floods, people avalanches, gas leaks, earthquakes, etc. Avoiding further human, economical or even environmental negative consequences.

A.2 Description of relevant impacts related to the project

This project mainly reinforces the security of public buildings, saving human costs by anticipating dangerous situations or managing them properly once they have already occurred.

The implemented software improves the efficiency of the previous one by swapping the used algorithm to carry out the evacuation from A star to Q-learning. This will allow a better performance, being able to apply it to more extensive and complex constructions.

It also avoids large investments in building deficient architecture designs, thanks to an accurate prediction of how the structure will unfold in emergency circumstances.

In addition, this paper provides different study lines to develop, as future work, a disaggregated evacuation based on the social profile of the building tenants. Taking into account their physical conditions, responsibilities, habits and other relevant characteristics, so that all social profiles are contemplated in the management of the evacuation: children, the elderly, disabled, etc.

In particular, this project does not affect directly to the environmental aspect, but it does propose a basis (algorithm) on which to raise a few applications of this nature. For example, providing optimization in the management of energy inside buildings.

A.3 Conclusion

This project contributes to the research and development of AI knowledge, giving an example of the enormous positive impact that artificial intelligence can have on our society.

APPENDIX B

Economic budget

This section exposes a detailed budget table of the whole project, covering aspects such as labor costs, material resources, general expenses, industrial benefit and others. Most part of the final budget is derived from the developer salary.

LABOR COST (direct cost)	Hours	Price/ Hour	Total
	300	16 €	4800,00 €

COST OF MATERIAL RESOURCES (direct cost)	purchase price	usage (in months)	Amortization (in years)	Total
Personal computer: SW IOS, RAM 8Gb, CPU I5.	1.500,00 €	6	5	150,00 €

TOTAL COST OF MATERIAL RESOURCES	150,00 €

GENERAL EXPENSES (indirect costs)	15%	CD	742,50 €
INDUSTRIAL BENEFIT	6%	CD+CI	341,55 €

FUNGIBLE MATERIAL

Print	100,00 €
Binding	300,00 €

SUBTOTAL BUDGET		6434,05 €
IVA	21%	1351,15 €

TOTAL BUDGET	7785,20 €

APPENDIX C

Neural Networks and Q-learning

Logically, as human beings, we do not need to have experienced a copy of each of the situations we are going to face, in order to know how to act successfully. We are able to extract separate aspects of previous experiences and extrapolate them to future ones, based on the similarity between them. For example, at some point in our past we learned that fire burns, and that we have to get away from it, wherever it is. This ability to classify a situation, and use the assets found in a similar one, allows us to save a huge amount of memory. If we analyse this factor from the machine learning perspective, it can be noticed the combination of two algorithms: the classification by supervised learning (CNN), and the experience stored in learning by reinforcement (Q-learning); every time we perform an action, we look for the previous one that gave us the greatest reward in a past similar circumstance.

The Convolutional Neural Network (CNN) is a very extended supervised learning technique used to classify inputs: such as text (1D tensor), images (2D tensor) or even video (3D tensor). To begin with, a CNN model is previously fed with sample-label verified matches. Afterwards, a sophisticated mathematical procedure based on node layers (filters), separates the most relevant features characterising the correspondent input. Finally, with all that stored information, the model is able to predict the label of a certain input (not used in the training process). This decision is based on the similitude of its features compared with the old, processed samples.

What if, this CNN classification is integrated before the agent choose the most suitable action, in such a way that generalizes its current situation to the most similar one stored in its q-table (experience). This fact would radically save the need of training the agent in thousands of different states and consequently, enable the assimilation of more complex environments.

This method already exists and is called Deep Q Learning. To easily understand this comparison consider the input images as game's screen-shots and the output predicted numbers (labels) as actions in a game, the result is illustrated at figure C.1. The design of this figure is taken from DeepMind, the artificial intelligence Google's branch that is nowadays taking the lead in this field.

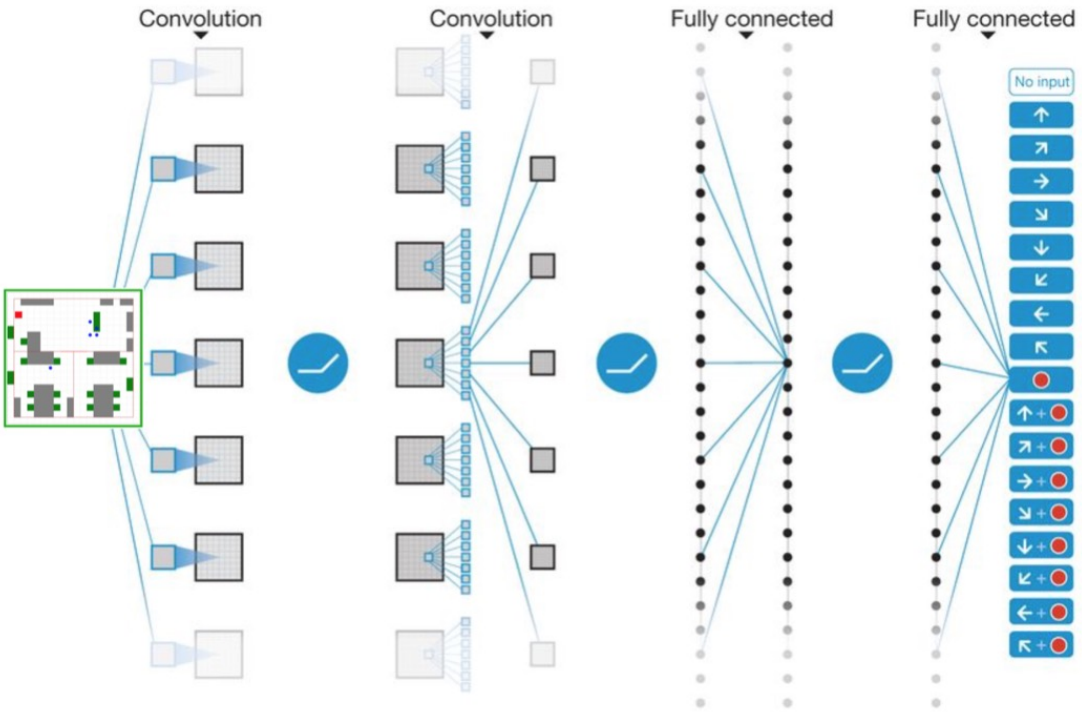

Figure C.1: DeepMind's Deep Q-learning sctructure applied to a SEBA's sample input image.

From a technical point of view, the communication between the frameworks and the neural network would be carried out through the matrix that represents the floor. Each of the elements placed inside the cells would be abstracted by numbers. In such a way that different situations could be classified and recorded by the layers of the neural network, extracting patterns and characteristics to carry out the prediction of the most suitable

action depending on the current context.

Bibliography

[1] Peter Abbeel. Markov decision processes and exact solution methods: Value iteration policy iteration linear programming, 2006.

[2] Mohammad Ashraf. Reinforcement learning demystified: Markov decision processes, 2017.

[3] Pablo Aznar. Design and Development of an Agent-based Social Simulation Visualization Tool for Indoor Crowd Analytics based on the library Three.js. Tfg, Universidad Politécnica de Madrid, June 2017.

[4] Richard Bellman. Dynamic programming, 1957.

[5] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems, 2002.

[6] Frantisek Duchonab. Path planning with modified a star algorithm for a mobile robot, 2014.

[7] GSI Eduardo Merino. Soba framework - documentation, 2015.

[8] Guillermo Fernández. Design and Implementation of an Agent-based Crowd Simulation Model for Evacuation of University Buildings Using Python. Tfg, Universidad Politécnica de Madrid, June 2017.

[9] Guzmán Gómez. Project's implementation: fire evacuation with q-learning. `https://github.com/GGP00`.

[10] D. Masad and J. Kazil. Mesa: Agent-based modeling framework in python 3+ — documentation, 2014.

[11] MHarea. Mathematics and computers in simulation, 2004.

[12] Henryk Michalewski. Playing atari on ram with deep q-learning, 2015.

[13] T.M. Mitchell R.S. Michalski, J.G. Carbonell. Machine learning: An artificial intelligence approach, 1983.

[14] Stuart J. Russell and Peter Norvig. Artificial intelligence: A modern approach.

[15] Thomas Simonini. Diving deeper into reinforcement learning with q-learning, 2017.

[16] Gerhard Weiss. Multiagent systems: A modern approach to distributed artificial intelligence.