UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Matematica "Tullio Levi-Civita"

Master Degree in Data Science

TESI DI LAUREA MAGISTRALE

A PROTOTYPE FOR INCREMENTAL LEARNING FINITE FACTORED PLANNING DOMAINS FROM CONTINUOUS PERCEPTIONS AND SOME CASE STUDIES

Relatore: SERAFINI LUCIANO

Laureanda: CLAUDIA FRACCA
Matricola: 1206037

20 Luglio 2020

Contents

Abstract

The project focuses on automated learning of planning domains. Automated planning often leverages on an abstract representation of the world called *planning domains*. A planning domain is described by a set of states that correspond to possible configurations of the environment, a set of actions, and a state-transition function between states, which describes the effects of actions on the environment. Planning domain are used by agents to develop their strategies on how to act in a given environment in order to achieve their goals. The construction of these planning domains is normally entrusted to the engineer who manually programs the agent. This work, however, requires human intervention for each new environment, while it would be desirable for an autonomous agent to be able to build a planning domain of the environment autonomously, even if it is in an unknown environment.

This master project takes place within the research activity carried on by the Fondazione Bruno Kessler (FBK) in Trento, in Data and Knowledge Management unit (DKM), with Luciano Serafini as internal tutor, and Paolo Traverso as external tutor. The aim of the research carried on by FBK is to use automatic methods to learn these models by carrying out actions, and observing their effects on the environment. In this research a framework for the automatic learning of planning domains was developed, based on perceptions, and within this framework an algorithm was formulated: the *Acting and Learning Planning domains* algorithm (ALP).

The specific objectives of this master thesis are to: (1) acquire specific knowledge in the field of automatic learning and planning; (2) acquire specific knowledge in the context of learning planning models; (3) extend the method developed in the research activity in FBK on states variables that are describes by a set of variables taking values over domains; (4) deal with the problem of learning entirely the state-transition function; (5) extend the ALP algorithm with a prediction part to predict the effects of the actions; (6) complete the state-transition function; (7) implement the algorithm with all the new parts, provide examples, and test this extension in some simulators previously created; (8) validate and analyse the results, and measure the goodness of the model.

Introduction

Automated learning and planning is a branch of artificial intelligence that aims to support the learning and planning activity by reasoning on conceptual models. Conceptual models mean abstract and formal representations of the environment and of the effects of actions, typically performed by intelligent agents, autonomous robots or vehicles [4].
An agent is any entity capable of interacting with its environment. An agent acting deliberately is motivated by some objectives. It performs one or several actions that are justifiable by sound reasoning with respect to this objective [2].
The intuition is that actions are executed in a given domain. They make the domain evolve and change its state. We use the word 'action' to refer to something that an agent can perform with its actuators, such as exerting a force, a motion or a communication, in order to make a change in its environment and its own state. For instance, in a robot navigation domain, an action moving the robot changes its position.

Automated planning is based on abstract conceptual models of the world, called *planning domains*. A planning domain is not an a priori definition of the agent and its environment. But instead, it is an approximation that must be a trade-off among several competing criteria: accuracy, computational performance, and understand-ability to users, see [5].
A planning domain is described by a set of states, which correspond to possible configurations of the world, a set of actions, and a state-transition function, which describes the effects of actions on the environment. This is an abstract representation of the real world, that can be useful for both conceptually and practically, because allows one to do formal reasoning and, also, to create a software solution that behaves accordingly.
Agents perceives the real world through sensors, that return data as a vector of continuous variables, called perception variables. It is part of the ability of the agent to fill the gap between the abstract model described by the planning domain and the real world perceived through the perception variables. The construction of these planning domains is normally entrusted to the engineer who manually programs the agent. This work, however, requires human intervention for each

new environment, while it would be desirable for an autonomous agent to be able to build a planning domain of the environment autonomously, even if it is in an unknown environment.

The work on planning and learning developed so far in the research activity carried on by the Fondazione Bruno Kessler (FBK) in the Data and Knowledge Management (DKM) unit, see [13] and [11], captures this idea of two levels. And in this research they provide a framework in which the agent can learn dynamically the new states of the planning domains, the mapping between abstract states. The perception from the real world is part of the planning domains and such mapping is learned and updated along the life of the agent [12].
They propose the *Acting and Learning Planning domains* algorithm (ALP), that interleaves planning, acting, and learning, and that builds a finite planning domains, and a perception function by executing actions and observing the effects in the real world through the perception variables. This algorithm relies on a perception function, which gives us a mapping between the state variables, i.e. variables used to represent our states, and the perception variables. To model the perceptions of the agent of the real world, ALP uses the perception function that returns the likelihood of observing continuous data, and it defines a criteria based again on the perception function to extend the set of states, i.e. when the likelihood is too low for all existing states then the model is extended. The perception variables are the only information about the real environment that is available to the agent during the learning.
The algorithm incrementally learns not only the values of the state variables, but also the description of the state-transition function, and the update of the perception function.
The learning of the planning domain can be defined by some parameters, that allow the agent to be more impulsive or more cautious in changing the model.
This schema provides the ability to learn, act, and plan not also in the routine situations but also in unexpected situations and adapt the model accordingly.

The current version of ALP comes with some limitations: to learn entirely the state-transition function, which describes the effects of actions on the environment, the agent has to physically perform the actions, and then it learns the transitions from the perceptions. This can become prohibitive in the case of a large number of states and actions, so we need a huge number of iterations to learn entirely the state-transition function. In this thesis I propose to use the artificial neural network to predict the perception variables given an action and starting perception variables. We decide to use online methods. So learning occurs during the execution phase. This kind of learning never stops, thus allowing the continuous correction, and improvement of an incorrect model and adaptation to changing environment characteristics. For every action we train a neural network that given

the current perception, it predicts the perceptions obtained after the execution of the action . We train this network as soon as we collect enough observations pairs (before and after the execution of the action). We then compare the prediction of the network with the real perception after the execution of the action. If the difference between the predicted perception and the true perception is small enough, then we use the prediction of the network to complete the planning domain with the missing transitions on the states.

The new contributions in this thesis are:

1. to extend the algorithm to states variables which are described by a set of variables taking values over domains;
2. to deal with the problem of learning entirely the state-transition function;
3. to extend the ALP algorithm with a prediction part to predict the effects of the actions on the environment;
4. to complete the transition function using the predictions, and update and correct this learned transition function along the iterations;
5. to implement the algorithm with all the new parts, provide examples, and test this extension in some simulators previously created;
6. to validate and analyse the results and measure the goodness of the learned model.

The thesis is composed by the following chapters:

In *Chapter 1*, we introduce two fundamental elements for the framework of this thesis: planning domains and perceptions variables. We also report two explanatory examples, which are used throughout the manuscript. In this chapter, these examples are used to better explain the basic concepts of planning domain, and the corresponding perception function. Finally, we define the problem of learning the planning domain, its challenge and its objectives.
Chapter 2 introduces the *Acting and Learning Planning domains* algorithm (ALP). We start describing one of the new contributions of this thesis: the extension of the algorithm considering the fact that the state variables are described by a set of variables taking values over domains. The key concept in which we dwell most are: (1) how update the planning domains and the perception function by introducing new states by extending the possible values of states variables, (2) how to adapt the perception function to fit the observed data, and (3) how to update the

state-transition function on the basis of the executed actions and observation of the effect in the real world through the sensor variables. Again we will see examples and experiments. Finally we define the problem which we will cover in the following chapters: the problem of learning entirely the transition function.
In *Chapter 3*, another new contribution is explained: prediction of actions effects. We describe the prediction part added to the ALP algorithm, we extend the algorithm using an artificial neural network that is used to predict the effects of an action in a given state. We will again implement and show in some case studies how it works, and we report results and examples.
Chapter 4 proposes a method to solve the problem of the completion of the transition function. Therefore the next step is to use the prediction to complete the state-transition function without doing the actions physically. We explain the problem and we propose a solution, we implement also this new part, provide examples and test this extension in some simulators previously created.
Finally, in *Chapter 5*, we estimate the quality of the model generated by our extended ALP algorithm. We evaluate the learned state space and the state-transition function. We provide examples and experiments that show this analysis.

Keywords: Planning, Online Learning, Knowledge Representation, Transition Systems, Perception.

Related Work

In recent literature, workshops and competitions, there is considerable interest in learning the planning systems. The acquisition of domains knowledge is widely recognised as a challenging bottleneck. In fact the Knowledge Engineering for Planning and Scheduling (KEPS) Workshop and the International Competition on Knowledge Engineering for Planning and Scheduling 2019 cover all aspects of knowledge engineering for AI planning and scheduling, and between the relevant areas of study there are: formulation of domains and problem descriptions, and methods and tools for the acquisition of domain knowledge. The automated learning of planning domains is a way to address this challenge. Indeed, most often, it is impossible to specify a complete and correct model of the world. Moreover, most of the times a model needs to be updated and adapted to a changing environment [5].

Several and different learning approaches have been proposed so far. Some works on domain model acquisition focus on the problem of learning action schema from collections of plans, see e.g. [6], [15]. Also in [11] and, [12] where they consider perceptions and the set of states not fixed a priori. And also other topics have been investigate like learning planning operators and domain models from plan examples and solution traces, see e.g. [8], and learning probabilistic planning operators, see e.g. [9]. In this work we use the approach and the solution proposed in [13].

Since the first days, artificial intelligence has been concerned with the problem of machine learning. There are many works that analyse the different ways machine learning can be used to improve automated planning problems, see [3] and [2]. Recent progress in deep learning and the availability of off-the-shelf tools such as TensorFlow [1] make it possible to learn highly accurate nonlinear deep neural networks with little prior knowledge of model structure [10]. We are interested the use of the machine learning to learn and complete the transition function.

In particular, [2] contains an overview of learning planning action models. In this work there is also a classification of these methods in two categories: online and offline methods. Online when learning occurs during the execution phase. The system can start learning as soon as the generation phase is complete. This kind of learning never stops, thus allowing the continuous correction and improvement of an incorrect model and adaptation to changing environment characteristics. On

the flip side, the cost of learning is added to the cost of planning, augmenting the time window in which execution occurs. Offline, instead, is one-shot learning exercise from traces. It allows a decoupling of the learning and planning phases, ensuring that the cost of learning is not added to that of planning. However, this one-time learning also means that in case of the injection of an ill defined domain, the planner may never be able to recover before the end of the planning and the beginning of the learning phase, thus staying blocked [2]. In our work we will focus online methods.

In the area of online learning and planning, there are recent works on learning from traces and action, see e.g. [6] and [10]. But in all these works the description of the traces is at the same level of abstraction as that of the resulting states, while in our approach we have that the observations are continuous and the states are discrete. Moreover, our approach shares some similarities with the work in [7] that proposes the deep planning network, a model-based agent that learns the environment dynamics from pixels and chooses actions through online planning in a compact latent space.

Chapter 1

Finite Factored Planning Domains and Continuous Perceptions

The descriptive models used by learning and planning systems are often called planning domains. It is important to underline that a planning domain is not an a priori definition of the agent and its environment. But instead, it is an approximation that must be a trade-off among several competing criteria: accuracy, computational performance, and understand-ability to users, see [5].

To start, in the first *Section 1.1*, we introduce the fundamental element for the framework of this thesis: the planning domain. We start defining the elements of a deterministic planning domain: the set of states, the set of actions, and the state-transition function.

Then, in the second *Section 1.2*, we formalize the concept of perception variables, i.e. continuous features that describes the state of the world. Successively, we define the perception function, i.e. a conditional probability distribution that mapping between state variables and perception variables. And we give the formal definition of extended planning domain.

In the third *Section 1.3*, we report a working example, which is used throughout the manuscript. In this chapter, the example is used to better explain the basic concepts of planning domain, and the corresponding perception function. The example simulates a robot that can move in a flat. This robot perceives the world through sensors, which return data as real variables. The robot is assumed to use odometers to sense its position, as coordinates, as well as the distances from the walls in the four directions (north, south, east and west).

Finally, in the last *Section 1.4*, we define the problem of learning the planning domain, its challenge and its objectives. And we give the key ideas behind the development of the ALP algorithm.

1.1 Planning Domains

A finite deterministic planning domains $\mathcal{D}$ is a triple $\mathcal{D} = \langle S, A, \gamma \rangle$.

Definition 1.1 (State-Transition System or Planning Domain, see [5])**.** *A state-transition system (also called a classical planning domain) is a triple* $\mathcal{D} = \langle S, A, \gamma \rangle$ *where:*

1. *S is a finite set of states in which the system may be.*
2. *A is a finite set of actions that the agent may perform.*
3. *γ is a function called the prediction function or state-transition function.*

Let's see in detail every single element of the state-transition system as defined in *Definition 1.1.*

State-Variable State Space

The state-variable state space S is a finite non-empty set of states. Each element $\mathbf{s}$ of this set is represented with a vector of values, i.e. a total assignment of *state variables* ranging over finite sets.
Let $\mathbf{V}$ be a vector of m state variables:

$$\mathbf{V} = (V_1, V_2, ..., V_i, ..., V_m) \tag{1.1}$$

where each V_i ranging over a finite set of values called *domain.*
Let:

$$D = \{D_1, D_2, ..., D_k\} \tag{1.2}$$

the set of non-empty finite k domains $D_1, ..., D_k$.
Let us define a function *dom* that assigns a domain $dom(V_i)$ to each variable V_i of $\mathbf{V}$.

$$\begin{aligned} dom\colon V &\longrightarrow D \\ V_i &\longmapsto dom(V_i) \end{aligned} \tag{1.3}$$

A state $\mathbf{s} \in S$ is a *total assignment*, i.e. a set of assignments that assigns a value $v_i \in dom(V_i)$ to every state variables V_i.
With these assignments we can denote the cross product of the domains of all the variables in $\mathbf{V}$. This cross product represents the set of the total assignments.

$$dom(\mathbf{V}) := \times_{V_i \in \mathbf{V}} dom(V_i) \tag{1.4}$$

Not every total assignment necessarily correspond to a state.
And as consequence the set of states S is a subset of the total assignments $dom(\mathbf{V})$:

$$S \subseteq dom(\mathbf{V}) \tag{1.5}$$

Definition 1.2 (State-Variable State Space, see [5])**.** *A state-variable state space is a set S of variable-assignments over some set of state variables V. Each variable-assignment in S is called a state of S.*

In the simplest case we can have that the function *dom* is a bijection, a one to one correspondence mapping of a set of state variables to a set of domains. In this case we have $k = m$.

Actions

We use the word action to refer to something that an agent does, such as exerting a force, a motion, a perception or a communication, in order to make a change in its environment and own state.
An agent is any entity capable of interacting with its environment. It performs one or several actions that are justifiable by valid reasoning with respect to some objectives. When an action is performed in a particular state, the state changes in response to that action. It can be the case that an action has no effect when executed in some states.
Let us define A a finite set of t actions and we denote this set as:

$$A = \{a_1, a_2, ..., a_t\} \tag{1.6}$$

We assume that the set of actions is the same for all states and each action can be executed at any time.

State-Transition Function

The state-transition function γ is a function that assigns a state in S given a state $\mathbf{s} \in S$ and an action $a \in A$.

$$\begin{aligned} \gamma &: S \times A \longrightarrow S \\ &(\mathbf{s}, a) \longmapsto \gamma(\mathbf{s}, a) \end{aligned} \tag{1.7}$$

The new state $\gamma(\mathbf{s}, a)$ is the result of the action a in the state $\mathbf{s}$. This function gives us the mapping between the pairs (state, action) for each state and for each action, with the state that is the result of applying the action to that state.

Model Assumptions

Here we define the assumptions that will be taken from now on, these model assumptions are called the classical planning assumptions, see [5].

1. **Finite environment**: the sets of states and actions have to be finite.
2. **Static environment**: in addition to requiring the sets of states and actions to be finite, we have the assumption that changes occur only in response to actions: if the agent does not act, then the current state remains unchanged. This excludes the possibility of actions by other agents, or events that are not due to any agent.
3. **No explicit time, no concurrency**: there is no explicit model of time (e.g., when to start performing an action, how long a state or action should last, or how to perform other actions concurrently). There is just a discrete sequence of states and actions $\mathbf{s}^{(0)}, a^{(1)}, \mathbf{s}^{(1)}, a^{(2)}, \mathbf{s}^{(2)}, \ldots$.
4. **Determinism, no uncertainty**: assumes that we can predict with certainty what state will be produced if an action a is performed in a state $\mathbf{s}$. The classical AI planning models assumes deterministic actions: any action taken in a state has at most one successor state.

1.2 Perception Function

The agent perceives the world around it through a vector of continuous variables, called *perception variables*.

$$\mathbf{X} = (X_1, X_2, ..., X_j, ..., X_n) \tag{1.8}$$

where $X_1, ..., X_n$ are n variables ranging over the real numbers.
And we can define a function, called *perception function*, that returns the likelihood to perceive these sensor variables given a state.

$$\begin{aligned} f &: \mathbb{R}^n \times dom(\mathbf{V}) \longrightarrow R^+ \\ (\mathbf{x}, \mathbf{s}) &\longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{n} p_{X_j}(x_j|V_{I_j} = v_{I_j}) \end{aligned} \tag{1.9}$$

where I_j is a subset of $\{1, ..., m\}$ for every perception variables X_j, and

$$V_{I_j} = \{V_i | i \in I_j\}$$

It can be define the *partial assignment* $V_{I_j} = v_{I_j}$ that assign to each variable $V \in \{V_i | i \in I_j\}$ a value $v \in dom(V)$.

The perception function is a probabilistic mapping between state variables and perception variables from the real environment. This function is a conditional probability distribution that computes the likelihood of perceiving some values of the sensor variables given an assignment to the state variables.
Intuitively, when the likelihood is high for a given state then we have an high probability to be in that state. On the other hand, when the likelihood is too low for all the existing states, we need to introduce new states by extending the possible values of the state variables, i.e. the domains.

In the simplest case we have that the mapping between the perception variables and the state variables is a bijection, in other words we have a one to one correspondence between these two sets of variables.
In this case the perception function become:

$$\begin{aligned} f &: \mathbb{R}^n \times dom(\mathbf{V}) \longrightarrow R^+ \\ &(\mathbf{x}, \mathbf{s}) \longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{n} p_{X_j}(x_j | V_j = v_j) \end{aligned} \tag{1.10}$$

with the number of state variables equal to the number of sensor variables.
Now we have all the elements to define the concept of *extended planning domains.*

Definition 1.3 (Extended Planning Domain, see [11])**.** *An extended planning domain, or a planning domain with perception function, is a pair* $\langle \mathcal{D}, f \rangle$*, where* $\mathcal{D}$ *is a planning domain* $\mathcal{D} = \langle S, A, \gamma \rangle$ *and* f *is a perception function on the state of* $\mathcal{D}$*.*

1.3 Case Studies: Description of Grid, and Grid with Walls Sensing

In the first two subsections: *Subsection 1.3.1*, and *Subsection 1.3.2*, we describe two simulators that we will call respectively 'Grid' and 'Grid with Walls Sensing' which will be used throughout the thesis to provide examples and results. Then in the third *subsection 1.3.3*, we describe an explanatory example in the 'Grid with Walls Sensing' to show the concepts formalized by now: the state transition system and the perception function. Finally, in the last *Subsection 1.3.4*, we underline some useful observations.

1.3.1 Description of the Grid Simulator

In this simulator we have a world represented by a grid. This grid has dimensions $m \times n$, the grid is divide into $m \times n$ cells.
In a real implementation the grid can represent a flat or a large room and these cells can represent the rooms of the flat or particular positions in the room.
Some of these cells can be closed, which represent in a real environment the possibility of a closed room or a pillar in the room. This means that there are cell in which is impossible to access. We will therefore have a defined number of free cells.

Example 1.4. *Let see an example in the Figure 1.1, in this case we have a* 4×4 *grid composed by 16 cells . This grid can represent a flat with sixteen rooms or a room with sixteen floor tiles. This grid is fenced with external walls.*

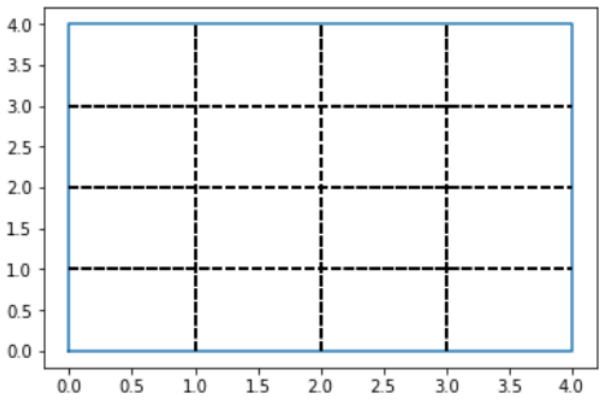

Figure 1.1: *A real world: the building has 16 cells and external walls.*

Inside this grid there is a robot. This robot can move freely in free cells. Given its initial position, the robot can move from one cell to another adjacent cell.
The actions it can carry out are four:

$$A = \{North, South, East, West\}$$

These actions will cause the robot to move one unit respectively in the action's direction. In any position it is in, therefore in any of the free cells, it can perform any of the four actions.

If it is not possible to move a unit in the chosen direction due to an external wall or a closed room, the robot will remain stationary in the previous position.

Example 1.5. *In our example we can suppose that some of these cells are closed, for example the robot have no access to two rooms, or there are some pillars in the room, and the robot has no access to that positions, see Figure 1.2.*

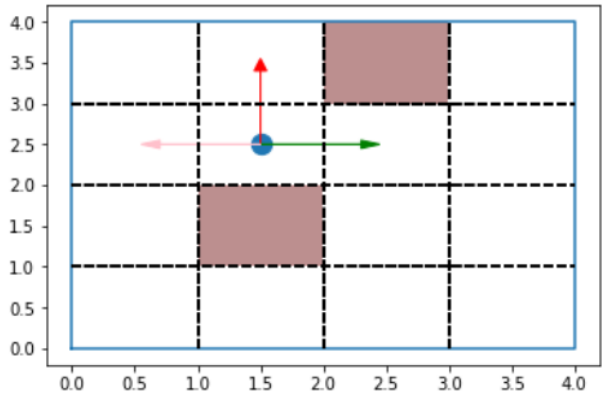

Figure 1.2: *A possible situation where we have a 4×4 grid composed by 16 cells but two of them are closed, we can imagine a room with two pillars. The robot's position in this case is in the coordinates (1.5,2.5) and the possible actions are: North, East and West. It can not move in the south direction. However, the action South can be selected and if it selects the action South it does not move.*

The robot perceives the surrounding world through position sensors. These sensors return two real numbers that correspond to the x and y coordinates of its position within the grid. Therefore the robot has the following two perception variables:

$$\mathbf{X} = (X_1, X_2)$$

where X_1 and X_2 are the x-coordinate and the y-coordinate of the position of the robot.
Clearly, the value of these perceptions variables returned by the sensors of the robot can be suffered by some small measurement errors or instruments sensitivity.

1.3.2 Description of the Grid with Walls Sensing Simulator

Let us describe an other simulator that we will use it throughout the thesis.
We call this simulator 'Grid with Walls Sensing' and we suppose to have a $m \times n$ grid composed by $m \cdot n$ cells and a robot, like the 'Grid' simulator described in the *Section 1.3.1.* We have again the four actions $A = \{N, S, E, W\}$. But in this case the robot perceives to be in a position of the building thought sensors (like odometers) and the distances from the nearest wall in the four directions: north, south, east and west.

The robot has the following six perception variables:

$$\mathbf{X} = (X_1, X_2, X_3, X_4, X_5, X_6)$$

where:

(i). X_1 and X_2 are the x-coordinate and the y-coordinate of the position of the robot;

(ii). X_3,X_4,X_5 and X_6 are the distances of the closest wall, in this order: north, south, east and finally west.

Example 1.6. *In this example we can see that respect to the case of the 'Grid' simulator, in this case we have also the sensors in the fourth directions that give us the distances from the nearest walls, see Figure 1.3.*

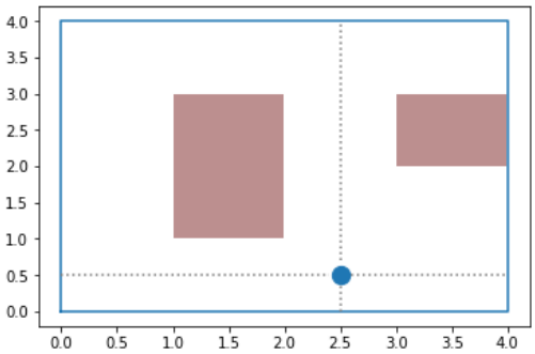

Figure 1.3: *In this configuration we have three closed cells and the position of the robot is symbolized by the blue dot in the coordinates (2.5,0.5). The robot not only perceives the positions but also the distances from the nearest walls in the four directions: north, south, east and west. In this case the perception values that the robot returns are (2.49,0.48,3.50,0.51,1.49,2.50).*

1.3.3 Example of Extended Planning Domain

In this example we describe all the elements that composed a planning domain for the configuration described in the *Figure 1.3*.

State variable state space We can suppose a one to one mapping between the perception variables and the state variables, so we have six state variables.

$$\mathbf{V} = (V_1, V_2, V_3, V_4, V_5, V_6)$$

Each variable with its domain:

$$D = \{D_1, D_2, D_3, D_4, D_5, D_6\} \tag{1.11}$$

The function *dom* that assigns a domains $dom(V_i)$ to each variable V_i of $\mathbf{V}$ is a bijection, so a one to one correspondence mapping of the set of state variable $\mathbf{V}$ to the set of domains D.
The state-variables state space S , in this case, is the set containing all the possible total assignments.

Actions The set of actions that the robot can performed are:
$A = \{North, South, East, West\}$. That, for simplicity, we will denote them as:

$$A = \{N, S, E, W\}$$

These actions can be performed by the robot in each state. In case of a closed cell or an external wall, the robot does not crash against the wall, but it perceives it and it does not move.

Perception Variables We have seen in the *Subsection 1.3.2* the meaning of the six perception variables. In particular, in the example in the *Figure 1.3* we have:

$$\mathbf{x}^{(1)} = (2.49, 0.48, 3.50, 0.51, 1.49, 2.50)$$

(i). $x_1^{(1)} \approx 2.49$ and $x_2^{(1)} \approx 0.48$ are the x-coordinate and the y-coordinate of the position of the robot;

(ii). $x_3^{(1)} \approx 3.50, x_4^{(1)} \approx 0.51, x_5^{(1)} \approx 1.49$ and $x_6^{(1)} \approx 2.50$ are the distances of the closest wall, in this order, north, south, east and finally west. In this case it does not perceived any internal walls, but only the external walls.

In a simple case we can presume that the robot generate an action randomly (a smarter strategy can take into account an *plan strategy* to reach some goal or an *exploration strategy*).
But, for now we can suppose that the robot selects an action at random, e.g. E, and move one step to east, and, using the sensors for the position and the sensors for the distances from the walls, it return another vector of perception variables, see *Figure 1.4*.

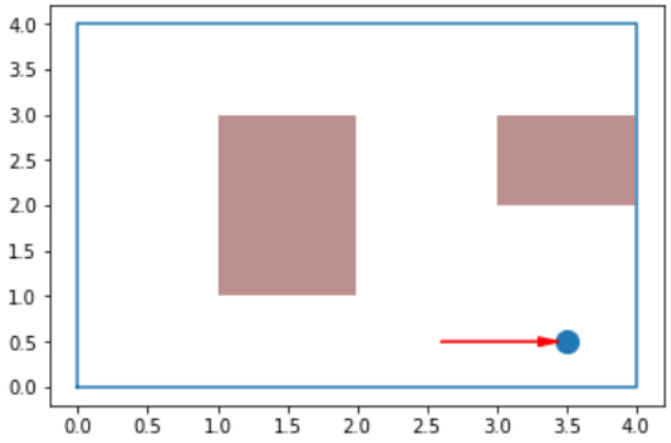

Figure 1.4: *In this configuration the robot executes the action E, represented with a red arrow. The position of the robot is symbolized by the blue dot in the coordinates (3.5,0.5). In this case the perception values that the robot return are (3.49,0.49,1.50,0.49,0.49,3.49).*

The observations that the robot returned after the execution of E are the following :

$$\mathbf{x}^{(2)} = (3.49, 0.49, 1.50, 0.49, 0.49, 3.49)$$

We can observe that, compared to before, we have:

(i). A different x-coordinate, because the robot is moving east approximately 1 unit;

(ii). The y-coordinate is more or less the same because the robot is moving approximately horizontally;

(iii). The south wall reader gives us more or less the same value because the distance from the south wall is the same;

(iv). The east and west wall sensing decrease and add one unit respectively;

(v). The north wall sense change from $x_3^{(1)} = 3.50$ to $x_3^{(2)} = 1.50$ because of the identification of the internal wall.

Perception Function As perception function we have the product of normal distributions:

$$\begin{aligned} f &: \mathbb{R}^6 \times dom(\mathbf{V}) \longrightarrow R^+ \\ (\mathbf{x}, \mathbf{s}) &\longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{6} N(x_j|\mu_{s_j}, \sigma_{s_j}^2) \end{aligned} \tag{1.12}$$

where $\mu_{s_j}, \sigma_{s_j}^2$ are the mean and the variance of the sensing previously stored for the specific value of this domain.

Transition Function Suppose that $\mathbf{s}^{(1)}$ is the state that represents the first position $\mathbf{x}^{(1)}$ of the robot and that state $\mathbf{s}^{(2)}$ represents the position $\mathbf{x}^{(2)}$. So in the transition function we will have this transition:

$$\mathbf{s}^{(2)} = \gamma(\mathbf{s}^{(1)}, E)$$

1.3.4 Observations

Observation 1.7. *Given the extended planning domain of the agent $\langle \mathcal{D}, f \rangle$, with $\mathcal{D} = \langle S, A, \gamma \rangle$ and f is a perception function, a good model is when we have coherence between the abstract model and the real world effects of an action perceived through the perceptions variables.*
Intuitively, if

$$\gamma(\boldsymbol{s}^{(1)}, a) = \boldsymbol{s}^{(2)}$$

then if the agent perceives to be in $\boldsymbol{s}^{(1)}$ and it performs a, then after the execution of a it will perceive to be in the state $\boldsymbol{s}^{(2)}$.
Following the example before, first of all the first observation that robot perceives is:

$$\boldsymbol{x}^{(1)} = (2.49, 0.48, 3.50, 0.51, 1.49, 2.50)$$

and suppose that the state that maximizes the likelihood of perceiving $\boldsymbol{x}^{(1)}$ is :

$$\boldsymbol{s}^{(1)} = (0, 0, 0, 0, 0, 0)$$

Then an action is generated randomly, let's suppose, following the example before, it is E.

The observation returned after the execution is:

$$\boldsymbol{x}^{(2)} = (3.49, 0.49, 1.50, 0.49, 0.49, 3.49)$$

According to its abstract model the agent will believe to be in the state:

$$\boldsymbol{s}^{(2)} = \gamma(\boldsymbol{s}^{(1)}, a) = \gamma((0, 0, 0, 0, 0, 0), E)$$

If instead, it perceives to be in a different state respect the prediction of the transiting function, then we have an incoherence between the real world and our model.

Observation 1.8. *There may be situations in which the robot perceives data which are not compatible with any of the states of its abstract model. For instance, the robot could be in an unknown states. Consider the examples that a closed room is opened, and the robot ends up to a different part of the building, so now the number of states are incomplete to describe this situation. Therefore, if we have a fixed set of states, these can be inadequate to represent the real world. But a dynamic set of states can have a key role, in fact if we extend and update the set of state when we understand that the actual set of states is not adequate to describe the new sensor variables, we are able to represent also unknown states and unexpected situations.*

1.4 Learning a Planning Domain

1.4.1 Problem Definition

In the recent literature, workshop and competition there is considerable interest in learning for planning systems, see in the section *Related Work*.

The acquisition of domains knowledge is widely recognised as a challenging bottleneck. In fact the Knowledge Engineering for Planning and Scheduling (KEPS) Workshop and International Competition on Knowledge Engineering for Planning and Scheduling 2019 said:

> Despite the progress in automated planning and scheduling systems, these systems still need to be fed by carefully engineered domain and problem descriptions, and fine tuned for particular domains and problems. Knowledge engineering for AI planning and scheduling deals with the acquisition, design, validation and maintenance of domain models, and the selection and optimization of appropriate machinery to work on them. These processes impact directly on the success of real-world planning and scheduling applications. The importance of knowledge engineering techniques is clearly demonstrated by a performance gap between domain-independent planners and planners exploiting domain-dependent knowledge.

The workshop covers all aspects of knowledge engineering for AI planning and scheduling, and between the relevant areas there are: formulation of domains and problem descriptions, and methods and tools for the acquisition of domain knowledge.

Learning by observations and by practice provides a new point in the space of approaches for problem solving and learning. There are many difficulties for such integration of learning and planning. The observations of an agent consist of: 1) the actions being executed, 2) the state in which each action is executed, and 3) the state resulting from the execution of each action.

The automated learning of planning domains is a way to address this challenge. Indeed, most often, it is impossible to specify a complete and correct model of the world. Moreover, most of the times a model needs to be update and adapted to a changing environment, see [5].

The aims are to learn a discrete deterministic planning domain $\langle \mathcal{D}, f \rangle$, with $\mathcal{D} = \langle S, A, \gamma \rangle$, starting from an empty planning domains. In particular learning method applies when the domains can be modelled with discrete actions and observable states.

We want an algorithm that builds a finite planning domains and a perception function by executing actions and observing the effects through the sensor vari-

ables. The only information about the real world that is available to the learning algorithm is provided by the perceptions variables. The algorithm not only has to learn the transition function and extend the set of states by extending the domain of some state variables, but also it has to learn how to update the perception function and the transition function. The challenges are:

- The needs to fill the gap between the real world and the abstract world: while an agent can conveniently plan at the abstract level, it perceives the world and acts in it through sensors and actuators that work with data in a continuous world, typically represented with variables on real numbers, see *Observation 1.7*;
- Deal with unexpected observations: there may be situations in which the agent perceives data which are not compatible with any of the states of its abstract model, see *Observation 1.8*.

1.4.2 Learning with ALP Algorithm

Most of the works in planning and learning, see in the section *Related Work*, assume that:

- the finite set of states of the planning domain is fixed once forever at design time;
- the correspondence between the abstract states and the observations is implicit and fixed at design time.

This is the case of most of the works on planning by reinforcement learning, which focus on learning and updating the transitions between states.
They support neither the learning of new states corresponding to unexpected situations the agent may encounter, nor the updating of the mapping between the perceptions represented with continuous variables and the abstract discrete model.
In many cases, however, having a fixed set of states and a fixed mapping between the perceived data and the abstract model is not adequate.
In fact, there may be situations in which the agent perceives data which are not compatible with any of the states of its abstract model.
For instance, we have see in the *Observation 1.8* of the *case study 1.3*, that the robot may end up in unknown and unexpected states of the world. Similarly, along its life, an agent could also revise its mapping between its abstract model and the real sensed data.
In general, the number of states and the mapping to perceptions may be not obvious at design time, and thus be incomplete or not adequate.
Therefore the objective are given formal framework in which:

- the agent can learn dynamically new states of the planning domain;
- the mapping between abstract states and the perception from the real world, represented by continuous variables, is part of the planning domain;
- such mapping is learned and updated along the life of the agent.

The *Acting and Learning Planning Domains* algorithm provide a formal framework to solve these objective. It gives a framework in which the agent can learn dynamically new states of the planning domain. Moreover, the mapping between abstract states and perceptions from the real world is part of the planning domain of the agent, and it is learned and updated along the life of the agent, it is also able to discover that the abstract model is not coherent with the real world, see [11]. We will describe this algorithm in *Chapter 2*.

Chapter 2

Incremental Learning Algorithm

In this chapter we present the *Acting and Learning Planning Domains* algorithm. Especially in the first *Section 2.1*, we describe in detail all the steps of the algorithm: the initialization, the inputs, and the pseudo-code for the core part. We analyze in detail the extension of the domains to introduce new states when the likelihood in under threshold, and how to update the perception function and the transition function.
Then in the second *Section 2.2*, we describe an example of the algorithm in a simulator. The example is used to better explain how the algorithm works, and to underline some important aspects and features.
Finally, in the last *Section 2.3*, we define the problem of learning the transition function, and we give the objectives for the next chapters.

2.1 Acting and Learning Planning Domains Algorithm (ALP)

To deal with the challenges and the objectives describe in the *sections 1.4.1* and *1.4.2*, the idea is to have a framework, that use the formal schema describe in *Chapter 1*, in which:

- it can be possible to model agent's perception of the real world by a perception function that returns the likelihood of observing some continuous data being in a state of the domain;

- describe an algorithm that interleaves planning, acting, and learning. It is able to discover that the abstract model is not coherent with the real world.

The general schema of ALP is:

Given an initialization, loop until a stopping condition holds:

1. *select an action;*
2. *execute the action by the agent and observe* $\boldsymbol{x}$*;*
3. *decide in which state it is based on maximum likelihood.*
   If the likelihood is too low for all the existing states, create a new state;
4. *update the transition function;*
5. *update the perception function.*

We will see in details in the next sections all these steps.

2.1.1 Initialization

Initial planning domain

ALP algorithm can start 'from scratch', that means from an empty planning domain. Another possibility is to start from a simple non-empty planning domains, that correspond to our prior knowledge about the reality.
For example, in a easy initial setting, we can have an initial non-empty planning domains where each variable takes value in non-empty domains. These domains can have only one value each: $D_1 = \{0\}, ..., D_k = \{0\}$.
In particular, from now on, throughout all this manuscript, we choose to start 'from scratch'. This choice is done without loss of generality: any other choice of initialization for the planning domain is possible.
Therefore, starting from an empty planning domain means starting with a state-transition system $\mathcal{D} = \langle S, A, \gamma \rangle$ where :

(i). S is an empty set of states: $S = \varnothing$.
And for each state variables V_i in $\mathbf{V}$ the domains are empty.

$$D_1 = \varnothing, ..., D_k = \varnothing$$

And the mapping between the set of state variables and the set of domains is given by a function *dom*.

(ii). A given finite set of actions $A = \{a_1, a_2, ..., a_t\}$;

(iii). And an empty transition function γ.

Initial perception function

The ALP algorithm also needs among its inputs an initial factored perception function defined for all variables assignment $V_{I_j} = v_{I_j}$, see *section 1.2*:

$$f\colon \mathbb{R}^n \times dom(\mathbf{V}) \longrightarrow R^+$$
$$(\mathbf{x}, \mathbf{s}) \longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{n} p_{X_j}(x_j|V_{I_j} = v_{I_j})$$

This perception function is defined as a factored conditional probability, i.e. product of p_{X_j}.
Let's suppose that each function p_{X_j} belongs to a parametric family of functions with parameters $\boldsymbol{\theta}_{X_j}$.

$$\boldsymbol{\theta}_{X_j} = (\theta^1_{X_j}, \theta^2_{X_j}, ..., \theta^{p_j}_{X_j})$$

For every partial assignment $V_{I_j} = v_{I_j}$ we have to setting the parameters $\boldsymbol{\theta}_{X_j}$ to some value $\boldsymbol{\theta}_{X_j, v_{I_j}}$.
In the case of empty initial planning domains we have no assignments, so in our case the initial factored perception function is empty, i.e. we don't have any setting of parameters.

We also need to define an initialiser p_{init,X_j} for every perception variable X_j. In fact since ALP can introduce new values in the domains of state variables, we need a method to initialise the parameters for perception function to these new values.

Inputs of the ALP algorithm:

Now we have all the elements to introduce the inputs for *Acting and Learning Planning domains* algorithm, see the pseudo-code in *Algorithm 1.*

This algorithm requires as input:

1. An initial planning domain $\mathcal{D} = \langle S, A, \gamma \rangle$, as described before;
2. An empty initial factored perception function f, as described before;
3. An initialiser p_{init,X_j} for every perception variable X_j, as described before;
4. A parameter *max_iter* that is the maximum number of iterations of our algorithm;
5. Other additional parameters in input:

   (i). $\alpha \in [0, 1]$ an update parameter for the transition function γ;

(ii). $\beta \in [0, 1]$ an update parameter for the perception function f;

(iii). $\epsilon \in [0, 1]$ a parameter for the minimum likelihood.

These parameters allow to the agent to define its behaviour, if it wants to have an approach more cautious or the other way around, more impulsive. And they express in some way how much the agent trusts in the structure of the model. In fact, each of these three parameters represent a key component of the model: α for the transitions, β for the perception function and ϵ for the states. We will see in detail the meaning of each parameter and how the setting of these three quantities influence the whole algorithm.

2.1.2 Description of the ALP Algorithm

Let us start describing the *Acting and Learning Planning domains* algorithm step by step. In the *Algorithm* 1 there is the frame of the structure.

input : initial planning domain: $\mathcal{D} = \langle S, A, \gamma \rangle$
input : initial perception function: f
input : perception initialization: p_{init,X_j}
input : parameters: α, β, ϵ
input : number of iterations: max_iter

$\mathcal{O} \leftarrow \{\}$ # *Empty history of observations* ;
$\mathcal{T} \leftarrow \{\}$ # *Empty history of transitions* ;
$\mathbf{x} \leftarrow$ sense() ;
current_state $\leftarrow$ best_current_state() ;
while *iter* $<$ *max_iter* **do**
  a $\leftarrow$select_action();
  execute_action(a) ;
  $\mathbf{x} \leftarrow$ sense();
  current_state $\leftarrow$ best_current_state() ;
  $\mathcal{O} \leftarrow$ bring up-to-date ;
  $\mathcal{T} \leftarrow$ bring up-to-date ;
  update_gamma();
  update_f();
  iter +=1;
end

Algorithm 1: Pseudo code for ALP

Before to start with the iterations of the algorithm we define two empty sets $\mathcal{O}$ and $\mathcal{T}$. These sets store the past observations and the history of the process and

they will be useful especially in the update of the transition function γ and in the update of the perception function f.
In detail we have $\mathcal{O}$ that is the empty history of observations. It will contain the states and the observations related to that state given by the sensing variables.

$$\begin{aligned}\mathcal{O} = \{&\mathbf{s}^{(1)} : (\mathbf{x}^{(11)}, \mathbf{x}^{(12)}, ..., \mathbf{x}^{(1h_1)}) \\ &... \ ..... \\ &\mathbf{s}^{(l)} : (\mathbf{x}^{(l1)}, \mathbf{x}^{(l2)}, ..., \mathbf{x}^{(lh_l)})\}\end{aligned}$$

We will denote with $\mathcal{O}(\mathbf{s})$ all the observations, i.e. the values of perception variables, observed in the past for the state $\mathbf{s}$.
And $\mathcal{T}$ is the empty history of transitions. It will contain the transitions:

$$(state, action, next\_state)$$

This means that if the agent perceived to be in a *state*, after performing the *action*, it had perceived to be in the *next_state*. And the transitions are stored with the number of times $N_{(state,action,next\_state)}$ which in the past it has performed that transitions.

$$\begin{aligned}\mathcal{T} = \{&(\mathbf{s}^{(1)}, a_t, \mathbf{s}^{(2)}) : N_{(\mathbf{s}^{(1)}, a_t, \mathbf{s}^{(2)})} \\ &..... \ ... \\ &(\mathbf{s}^{(l)}, a_t, \mathbf{s}^{(h)}) : N_{(\mathbf{s}^{(l)}, a_t, \mathbf{s}^{(h)})}\}\end{aligned}$$

Before to enter in the cycle *while*, see *Algorithm 1*, we have to select the initial state, so the agent, after being turned on, answers to the question: *where am I?*. To do that, it starts perceiving the world around it: the agent executes the function *sense*() that returns a vector of values of the perception variables.
After that ALP,considering that we start from an empty planning domain, has to extend the domains to define the first state.
To do that, the function *best_current_state*() is called. This function decides in which state the agent is, based on the maximum likelihood. If the likelihood is too low for all existing states or we don't have any state, the function create a new state. We will see in detail how work this function in the *section 2.1.3*.
Now we are ready to enter in the *while* cycle, where ALP iteratively refines the planning domain and the perception function by executing the actions, chosen in the function *select_action*(), and observing the effects of the execution of this action in the real world thought the sensing of the perception variables.
The function *select_action*() selecta an action in the set $A = \{a_1, a_2, ..., a_t\}$.
In a simple implementation this function selects an action at random. Other strategies can take into account a "plan strategy" to reach some goal or an "explore

strategy", that using the information that the agent has already learned, to explore the part of domains that still requires more learning.
After the selection of the action, ALP call the function $execute\_action(a)$, so the agent executes the action physically in the real environment.
After the execution of the action, the agent return the observations, through the function $sense()$, i.e. it perceives the world thought sensors that return data represented with variables on real numbers.
In order to determine the states that are the candidates to be the next states ALP call the function $best\_current\_state()$ that return the state for which the likelihood of observing this perception variables is higher than a certain threshold. If there are no good states, i.e. when the likelihood is too low for all the existing states, a new state has to be created. So the algorithm will extend the domains of some of the state variables. We see this function in detail in the next *section 2.1.3*.
After that ALP extends the transition history $\mathcal{T}$ adding the last transition:

$$\mathcal{T} \leftarrow \mathcal{T} \cup (last\_state, action, current\_state)$$

and it extends also the observations history $\mathcal{O}$ adding the last new observation:

$$\mathcal{O} \leftarrow \mathcal{O} \cup (current\_state, \mathbf{x})$$

After that ALP update the transition function, using the function $update\_gamma()$ and the perception function, $update\_f()$. These functions update the transition function γ and the perception function f using respectively the data available in the sets $\mathcal{T}$ and $\mathcal{O}$. The update functions take into account the current model, the observations and the transitions of the past and the new data just observed.
These functions can be setted in several different ways, depending on the agent strategy. It can be more cautious or more impulsive. In the first case be cautions means that changes are made, respectively, if there is a certain number of evidences from acting and perceiving the real world. Instead be impulsive in when just one unexpected evidence is enough to change the model.
This difference of strategies is regulated by some parameters, for the function $update\_f()$ the update parameter is β and for the function $update\_gamma()$ is α. These parameters are real number in the interval $[0, 1]$. We will describe in detail these two functions in the *sections 2.1.4* and *2.1.5*.

After that we adding an iteration in the iteration counter *iter* and we perform the cycle *while* until the number of execution is equal of the maximum number of execution max_iter, given between the inputs of the algorithm.

2.1.3 Minimum Likelihood and Extension of Domains

In order to determine the states that are the candidates to be the next states ALP call the function *best_current_state*() that return the state for which the likelihood of observing this perception variables is higher than a certain threshold. If there are no good states, i.e. when the likelihood is too low for all the existing states, a new state has to be created. So the algorithm will extend the domains of some of the state variables.
The pseudo code of the function *best_current_state*() is in the picture *Algorithm* 2. Let us explain it in detail.

input : a vector of perception variables: **x**
output: the best candidate to be the next state: *best_state*

if $S \neq \varnothing$ **then**
  best_state ← select_best_state() ;
  var_bad ← select variables under the min likelihood ;
  if *not var_bad* **then return** best_state;
end
if $S = \varnothing$ **then**
  var_bad ← all variables
end
extend_domains(var_bad) ;
best_state ← select_best_state() ;
return best_state

Algorithm 2: Pseudo code for *best_current_state*()

First of all the agent perceives the world around its thought a vector of values of some continuous variables:

$$\mathbf{x} = (x_1, x_2, ..., x_j, ..., x_n)$$

Then it call the function *best_current_state()*, and it can be in two possible situations: in the initial case with an empty set of states S or in the case of non-empty set of states S.

Initial case with an empty set of states

If we are at the beginning, at the first iteration $iter = 0$ of the *Algorithm 1* before the *while* cycle, we have all the sets of the domains empty. Therefore, we don't have any state in S. So when we call the function *best_current_state*() to find out a good state for the previous sense we do not have any choice and we inevitably

have to extend all the domains. Following the *Algorithm* 2, we enter in the second if of the function *best_current_state*().

In this case, we initialise all perception variables as bad variables, i.e. variables that we need to extend their domain because we don't have a good value for the domains.
Therefore, if $S = \varnothing$ then the variables that we set as variables under the threshold are by default:

$$var\_bad = \{X_1, X_2, ..., X_n\}$$

Case of non empty set of states

Otherwise, if the set of state is not empty, $S \neq \varnothing$, we are in the first if of the *Algorithm* 2, and we have to compute the likelihood.
If the likelihood is high then we have an high probability to be in that state, otherwise when the likelihood is too low for all the existing states we need to extend the domain of some variables.
To do that we calculate the perception function for the observation $\mathbf{x}$ for all the states and select the best one, i.e. the one that maximize the perception function.

$$\begin{aligned} f &: \mathbb{R}^n \times dom(\mathbf{V}) \longrightarrow R^+ \\ &(\mathbf{x}, \mathbf{s}) \longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{n} p_{X_j}(x_j|V_{I_j} = v_{I_j}) \end{aligned} \tag{2.1}$$

Then we check if there are some variables under the minimum likelihood.
Intuitively we want to select a set of states that are candidates to be the next state, for which the likelihood of observing x_j is higher than a certain threshold called minimum likelihood that depends also by a parameter ϵ.
This step can be done twice depending of our definition of the set of the states. Let us remember that:

$$S \subseteq dom(\mathbf{V}) \tag{2.2}$$

where:

$$dom(\mathbf{V}) := \times_{V_i \in \mathbf{V}} dom(V_i) \tag{2.3}$$

is the set of total assignments. In general not every total assignment correspond to a state. Therefore, before to extend the domains, ALP considers the assignments which are not in the set of states i.e. $dom(\mathbf{V}) \setminus S$.

Extend the Domains

In the case we have that the set of perception variables X_j, where the likelihood to perceived the value x_j is under the threshold then we need to extend the possible assignments to variable by extending their domains. This is performed in the function *extend_domains*(*var_bad*) which extends the domain of one or more state variables. For each variable X_j we select its domains.

$$D_{I_j} = \{dom(V) | V \in V_{I_j}\} \tag{2.4}$$

These are the domains to be extended with a new value.

Since we want to introduce a minimum number of values we choose to minimize the minimal hitting set between these sets D_{I_j}.

Definition 2.1 (Hitting Set)**.** *Given a family of sets* $\{B_i\}_{i=0}^n$*. A set A is said to be an hitting set of the family of sets* $\{B_i\}_{i=0}^n$ *if:*

$$A \cap B_i \neq \varnothing \qquad \forall i \in \{1, ..., n\} \tag{2.5}$$

Definition 2.2 (Minimal Hitting Set)**.** *Given a family of sets* $\{B_i\}_{i=0}^n$*. A set A is said to be a minimal hitting set of the family of sets* $\{B_i\}_{i=0}^n$ *if A is an hitting set:*

$$A \cap B_i \neq \varnothing \qquad \forall i \in \{1, ..., n\} \tag{2.6}$$

and not exist another hitting set C for the family of sets $\{B_i\}_{i=0}^n$ *such that:*

$$\nexists C \text{ hitting set} : |C| \leq |A| \tag{2.7}$$

Then each domain in the minimal hitting set of the set of family $\{D_{I_j}\}_{V_j \in var\_bad}$ is extended with a new value.

After that ALP extend the domains it has to initialise the perception function for the new variables without the parameters for the perception function. Using the initialiser p_{init,X_j} for every perception variable X_j i.e. a method to initialise the parameters for perception function to these new values.

Example 2.3. *Let us suppose that the set of perception variables that are under the threshold are* $\{X_1, X_2\}$ *and* $V_{I_1} = \{V_1, V_2\}$ *and* $V_{I_2} = \{V_2\}$*. The minimal hitting set in this case is* $D_H = \{dom(V_2)\}$*. Therefore we will extend* $dom(V_2)$ *with a new value* $|dom(V_2)|$*, resulting a set of* $|dom(V_2)| + 1$ *elements:* $\{0, 1, 2, ..., |dom(V_2)|\}$

The introduction of the new values for the state variables and the initialization of the perception function guarantee that now we have a good candidate to be the next state i.e. it has a likelihood above the threshold. Then ALP selects that state and the function *best_current_state*() return it.

Update the minimum likelihood to introduce new states

At each iteration we update the minimum likelihood

$$min\_likelihood = min\_likelihood \cdot (1 - \epsilon) \cdot e^{-\epsilon_d \cdot iter}$$

where the parameter $\epsilon \in [0, 1]$.
In the extreme case:

(i). $\epsilon = 1$ then $min\_likelihood = 0$ so we select all states in S;

(ii). ϵ higher, lower probability to introduce new states;

(iii). ϵ lower, higher probability to introduce new states.

Intuitively, ϵ express how much we believe that the set of states learned so far are sufficient for the planning domains to model the real world. And this is why we have also the term

$$e^{-\epsilon_d \cdot iter}$$

where $\epsilon_d \in [0, 1]$ is an input parameter, and it stands for epsilon decay.
In fact this term depends on the number of iterations, so as the number of iterations increase than the exponential go to zero. So, multiplied to the $(1 - \epsilon) \cdot min\_likelihood$ decrease this quantity as the iteration increases. Intuitively, express the fact that as the iterations increases we believe incrementally more and more that the set of states learned so far is sufficient for the planning domains to model the real world.
The variables that have the likelihood low than this minimum likelihood are insert in the list of bad variables.
If this list is empty then we don't have to extend the domain for some variable and we have a good candidate to be the next state given our perception variables. So, in the Algorithm 2 we return this best state.
Otherwise, if we have some variables with the likelihood too low for all the existing states we need to extend the domain.

2.1.4 Update the Perception Function

The idea is to update the perception function to maximize the likelihood of the entire set of observations extended with the new observation. The update of the perception function is based on the current perception function:

$$\begin{aligned} f &: \mathbb{R}^n \times dom(\mathbf{V}) \longrightarrow R^+ \\ &(\mathbf{x}, \mathbf{s}) \longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{n} p_{X_j}(x_j|V_{I_j} = v_{I_j}) \end{aligned} \tag{2.8}$$

where I_j is a subset of $\{1, ..., m\}$ and

$$V_{I_j} = \{V_i | i \in I_j\}$$

and the set of history of the observations, $\mathcal{O}$.

This perception function is defined as a factored conditional probability, i.e. product of p_{X_j}. Each function p_{X_j} belongs to a parametric family of functions with parameters $\boldsymbol{\theta}_{X_j}$.

Example 2.4. *For example, let suppose that we have a vector of two perception variables,* $\boldsymbol{X} = (X_1, X_2)$ *and the perception function is factored as follows:*

$$p(\boldsymbol{x}|\boldsymbol{s}) = p_{X_1}(x_1|V_{I_1} = v_{I_1}) \cdot p_{X_2}(x_2|V_{I_2} = v_{I_2})$$

where p_{X_1} *is a Gaussian distribution with parameters:*

$$\boldsymbol{\theta}_{X_1} = (\mu_{X_1}, \sigma^2_{X_1})$$

where the parameters are the mean and the variance of the normal distribution. And p_{X_2} *is an exponential distribution with parameter:*

$$\boldsymbol{\theta}_{X_2} = (\lambda_{X_2})$$

where λ *is the rate parameter of the exponential distribution.*

For every partial assignment $V_{I_j} = v_{I_j}$ we have to setting the parameters $\boldsymbol{\theta}_{X_j}$ to some value $\boldsymbol{\theta}_{X_j, v_{I_j}}$.

Example 2.5. *Following the previous example with the normal and the exponential distributions, suppose that the variable* X_1 *depends on* $V_{I_1} = \{V_1\}$ *and the variable* X_2 *depends on* $V_{I_2} = \{V_1, V_2\}$.
For every value $d \in dom(V_1)$*, it has to be set the values :*

$$\boldsymbol{\theta}_{X_1,d} = (\mu_{X_1,d}, \sigma^2_{X_1,d})$$

that are the mean and the variance of

$$p_{X_1}(x_1|V_1 = d) = N(x_1|\mu_{X_1,d}, \sigma^2_{X_1,d})$$

And for every value $d \in dom(V_1)$ *and* $p \in dom(V_2)$*, it has to be set the values:*

$$\boldsymbol{\theta}_{X_2,d,p} = (\lambda_{X_2,d,p})$$

that is the rate parameter of:

$$p_{X_2}(x_2|V_1 = d, V_2 = p) = Exp(x_2|\lambda_{X_2,d,p})$$

Given a new observation $\mathbf{x}^{(k+1)}$ about a state $\mathbf{s}$ in S and the set of previous observations about that state $\mathbf{s}$ in $\mathcal{O}$:

$$\mathcal{O}(\mathbf{s}) = (\mathbf{x}^{(1)}, ..., \mathbf{x}^{(k)})$$

ALP initializes these parameters using the observations and then it has to update the values of all the parameters $\boldsymbol{\theta}_{X_j,\mathbf{s}}$ when a new observation about a state $\mathbf{s}$ is given.

The update equation, implemented in the *update_f*() function, is defined as follows:

$$\boldsymbol{\theta}'_{\mathbf{X},\mathbf{s}} = \beta \cdot \boldsymbol{\theta}_{\mathbf{X},\mathbf{s}} + (1 - \beta) \cdot argmax_{\boldsymbol{\theta}'} \mathcal{L}(\boldsymbol{\theta}', \mathbf{x}^{(1)}, ..., \mathbf{x}^{(k+1)}|\mathbf{s}) \tag{2.9}$$

where $\mathcal{L}$ is the maximum likelihood function.
Intuitively, with this equation is defined the new parameters of the updated perception function for a state $\mathbf{s}$ as a convex combination, based on the parameter β. This convex combination is between the old parameters and the likelihood of the all observations.

$$\mathcal{L}(\boldsymbol{\theta}', \mathbf{x}^{(1)}, ..., \mathbf{x}^{(k+1)}|\mathbf{s}) = \prod_{t=1}^{k+1} f(\mathbf{x}^{(t)}|\boldsymbol{\theta}', \mathbf{s}) \tag{2.10}$$

Using the maximum likelihood principle we have that:

$$\mathcal{L}(\boldsymbol{\theta}_{ML}, \mathbf{x}^{(1)}, ..., \mathbf{x}^{(k+1)}|\mathbf{s}) = argmax_{\boldsymbol{\theta}'} \mathcal{L}(\boldsymbol{\theta}', \mathbf{x}^{(1)}, ..., \mathbf{x}^{(k+1)}|\mathbf{s}) \tag{2.11}$$

The agent can be more or less cautious in the revision of the perception function. This in expressed by the parameter $\beta \in [0, 1]$.

(i). $\beta = 0$ the agent update always in fact the update equation become:

$$\boldsymbol{\theta}'_{\mathbf{X},\mathbf{s}} = argmax_{\boldsymbol{\theta}'} \mathcal{L}(\boldsymbol{\theta}', \mathbf{x}^{(1)}, ..., \mathbf{x}^{(k+1)}|\mathbf{s}) \tag{2.12}$$

(ii). $\beta = 1$ the agent never update:

$$\boldsymbol{\theta}'_{\mathbf{X},\mathbf{s}} = 1 \cdot \boldsymbol{\theta}_{\mathbf{X},\mathbf{s}} \tag{2.13}$$

(iii). the highter the values of β the more cautious in the agent in the revision.

Due to the fact that we suppose to have a factorized perception function:

$$f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{n} p_{X_j}(x_j|V_{I_j} = v_{I_j})$$

we can separately update each parameters $\boldsymbol{\theta}_{X_j}$:

$$\boldsymbol{\theta}'_{X_j,\mathbf{s}} = \beta \cdot \boldsymbol{\theta}_{X_j,\mathbf{s}} + (1 - \beta) \cdot argmax_{\boldsymbol{\theta}'_{X_j}} \prod_{t=1}^{k+1} p_{X_j}(x_j^{(t)}|\boldsymbol{\theta}'_{X_j})$$

Example 2.6. *Following the previous example with the normal distribution for X_1, we have that:*

$$p_{X_1}(x_1|V_1 = d) = N(x_1|\mu_{X_1,d}, \sigma^2_{X_1,d})$$

Therefore, to calculate the update equation we have to calculate the likelihood function and then maximize it.

$$\begin{aligned}\prod_{t=1}^{k+1} p_{X_1}(x_1^{(t)}|\boldsymbol{\theta}'_{X_1}) =& \prod_{t=1}^{k+1} N(x_1^{(t)}|\mu_{X_1}, \sigma^2_{X_1}) = \\ =& \prod_{t=1}^{k+1} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_1^{(t)}-\mu)^2}{2\sigma^2}} = \\ =& (2\pi\sigma^2)^{-\frac{k+1}{2}} e^{-\sum_{t=1}^{k+1} \frac{(x_1^{(t)}-\mu)^2}{2\sigma^2}}\end{aligned}$$

Now we have to calculate :

$$argmax((2\pi\sigma^2)^{-\frac{k+1}{2}} e^{-\sum_{t=1}^{k+1} \frac{(x_1^{(t)}-\mu)^2}{2\sigma^2}})$$

And after some calculation we have that:

$$\mu_{ML} = \frac{\sum_{i=1}^{k+1} x_1^{(t)}}{k+1} \tag{2.14}$$

$$\sigma_{ML} = \frac{\sum_{i=1}^{k+1} (x_1^{(t)} - \overline{X}_1)^2}{k+1} \tag{2.15}$$

where $\overline{X}_1 = \frac{\sum_{i=1}^{k+1} x_1^{(t)}}{k+1}$

2.1.5 Update the Transition Function

Given the extended planning domain of the agent $\langle \mathcal{D}, f \rangle$, with $\mathcal{D} = \langle S, A, \gamma \rangle$ and f is a perception function, a good model is when we have coherence between the abstract model and the real world effects of an action perceived through the perceptions variables.
Intuitively, if

$$\gamma(\mathbf{s}^{(1)}, a) = \mathbf{s}^{(2)}$$

then if the agent perceives to be in $\mathbf{s}^{(1)}$ and it performs a, then after the execution of a it will perceive to be in the state $\mathbf{s}^{(2)}$.
But if $\mathbf{s}$ is the state that maximises the likelihood and we have that it is different from the state predicted by the transition function of our planning domain:

$$\mathbf{s} \neq \gamma(\mathbf{s}^{(1)}, a) = \mathbf{s}^{(2)}$$

Then we have to deal with this discrepancy and decide to revise the transition function γ or not.
Since our domain is deterministic i.e. the transition function must lead to an single state, then if we are in the case that there is a discrepancy between the state predicted by the model and the state given by the analysis of the sensors, we can or change the transition function with the new state or leave it as it is.

The *update_gamma*() function decides whether and not to update the transition function.

The update of the transition function is based on the current transition function and the set of history of the transitions $\mathcal{T}$.

The update equation, implemented in the *update_gamma*() function, is defined as follows:

$$\alpha \cdot \mathbb{1}_{\mathbf{s}'=\gamma(\mathbf{s}^{(1)},a)} + (1-\alpha) \cdot |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s}^{'})\}| \tag{2.16}$$

and we select the state that maximize this quantity.

Considering that we have to choice, or change the new state with the new state $\mathbf{s}$ or leave the state that is predicted by the transition function $\mathbf{s}^{(2)}$, we can consider only these two cases.
For the state $\mathbf{s}^{(2)}$ the formula 2.14 become:

$$\alpha + (1-\alpha) \cdot |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s}^{(2)})\}|$$

And for the new state $\mathbf{s}$ the formula 2.16 become:

$$(1-\alpha) \cdot |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s})\}|$$

We want to select the state that maximize these quantities:

$$(1-\alpha) \cdot |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s})\}| \geq \alpha + (1-\alpha) \cdot |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s}^{(2)})\}| \tag{2.17}$$

After some calculation 2.15 become:

$$|\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s})\}| - |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s}^{(2)})\}| \geq \frac{\alpha}{1-\alpha} \tag{2.18}$$

The agent can be more or less cautious or impulsive in the revision of the transition function. This in expressed by the parameter $\alpha \in [0, 1]$.

(i). $\alpha = 0$ we are extremely impulsive, we do not trust our model and just one evidence makes us to change the model.

$$1 \cdot |\{i|T_i = (\mathbf{s}^{(1)}, a, \mathbf{s}^{'})\}| \tag{2.19}$$

(ii). $\alpha = 1$ we are cautious, we believe in the transition function of our model and we never change it.

$$1 \cdot \mathbb{1}_{\mathbf{s}'=\gamma(\mathbf{s}^{(1)},a)} \tag{2.20}$$

(iii). $\alpha \in (0, 1)$ we are in the intermediate cases, depending on the values, decreasing α we are more likely to change our model.

2.2 Case Study: Learning Planning Domain on Grid with Walls Sensing

Let us now show how the algorithm works in an explanatory example that shows the potentiality and the limitations of this procedure.
First of all, we show how the ALP works in the framework of the simulator 'Grid with Walls Sensing' described in *Section 1.3.2*. We first describe how this procedure learns new states by extending domains of the state variables, and then how update the perception function and the transition function.
Let start from scratch, i.e. from the simplest planning domains described in the *Section 1.3*:

(i). S is an empty set of states.
Let $\mathbf{V}$ be a vector of 6 state variables ranging over his empty domains:

$$D_1 = \varnothing, D_2 = \varnothing, ..., D_6 = \varnothing$$

As a consequence the set of the total assignments is empty.

$$dom(V) = \times_{V_i \in \mathbf{V}} dom(V_i) = \varnothing$$

(ii). A finite set of action, $A = \{N, S, E, W\}$;

(iii). An empty transition function γ;

(iv). And the initial empty perception function f, described in (1.9).

The robot starts from this empty initial setting and the final aim is to learn the planning domain using the only information available from the real world, i.e. the perception variables:

$$\mathbf{X} = (X_1, X_2, X_3, X_4, X_5, X_6)$$

Let's start describing step by step the first iterations of the algorithm in this case. We suppose to start in the situation described in the *Figure 2.1*.

1. The first step of ALP is *sense*(). Let suppose that the first observations that the robot returned are:

$$\mathbf{x}^{(1)} = (2.49, 0.48, 3.50, 0.51, 1.49, 2.50)$$

2. Then ALP search for the best state through the function *best_current_state*($\mathbf{x}$). In this case, we are in the first iteration, $iter = 0$, so the set of states is empty,

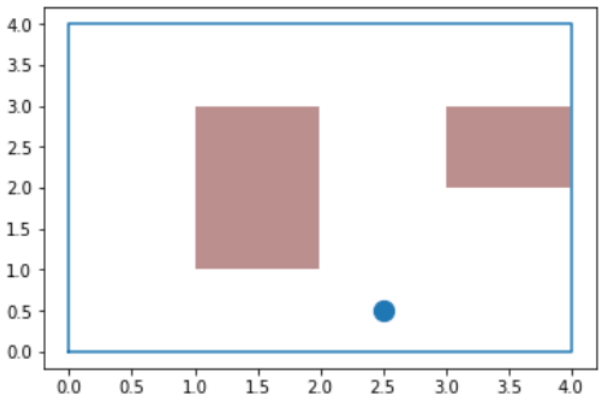

Figure 2.1: *In this configuration we have three closed cells and the position of the robot is symbolized by the blue dot in the coordinates (2.5,0.5). The robot not only perceives the positions but also the distances from the nearest walls in the four directions: north, south, east and west. In this case the perception values that the robot returns are (2.49,0.48,3.50,0.51,1.49,2.50).*

$S = \varnothing$.
As consequence, all the variables are considered under level, i.e. they have the perception function under the threshold, see *Algorithm 2*.

$$var\_bad = \{X_1, X_2, X_3, X_4, X_5, X_6\}$$

ALP has to extend all the domains and has to initialize the perception function for these values.

3. ALP generates therefore a new state by extending the domains of state variables, see function $extend\_domains(var\_bad)$ in the *Algorithm 2*.
All the domains are therefore extended with a new value.

$$D_1 = \{0\}, D_2 = \{0\}, ..., D_6 = \{0\}$$

As a consequence we have a state in S:

$$S = \{(0, 0, 0, 0, 0, 0)\}$$

Now ALP has to extends the perception function for the new assignments.

$$f : \mathbb{R}^6 \times dom(\mathbf{V}) \longrightarrow R^+$$
$$(\mathbf{x}, \mathbf{s}) \longmapsto f(\mathbf{x}, \mathbf{s}) = p(\mathbf{x}|\mathbf{s}) = \prod_{j=1}^{6} N(x_j|\mu_{s_j}, \sigma_{s_j}) \tag{2.21}$$

where μ_{s_j}, σ_{s_j} are the mean and the variance of the sensing previously stored for the specific value of this domain.
ALP initializes the mean and the variance using a set of observations X_{obs}. To do that we need other observations. So we call *sense*() many times as a parameter given in input, i.e. $|X_{obs}|$. In this case the method to initialise the parameter $\boldsymbol{\theta}_{Xj}$ given by the initializer p_{init,X_j} is to calculate the mean and the variance for the observations X_{obs}.

$$\mu_{s_j} = mean\{x_j | \forall \mathbf{x} \in X_{obs}\} = \frac{1}{|X_{obs}|} \sum_{\mathbf{x} \in X_{obs}} x_j$$

$$\sigma_{s_j} = var\{x_j | \forall \mathbf{x} \in X_{obs}\} = \frac{1}{|X_{obs}|} \sum_{\mathbf{x} \in X_{obs}} (x_j - \mu_{s_j})^2$$

We also calculate the minimum likelihood as

$$min\_likelihood_j = min\{N(x_j | \mu_{s_j}, \sigma_{s_j}) | \forall \mathbf{x} \in X_{obs}\}$$

Intuitively, considering that the values of perceptions variables returned by the sensors of the robot can be suffered by some small measurement errors or instrument sensitivity, we consider a set of observation and we presume that are normally distributed in a neighborhood of the real position of the robot.

4. Clearly, if we calculate the perception function f for all the elements in the domains we have that all the element that we have just add are greater than the minimum likelihood. So the state selected is (0,0,0,0,0,0) and we store in the history of observations $\mathcal{O}$ the state and the set of observations $X_{obs}+$ $\mathbf{x}^{(1)}$.

5. The current state is set to $(0, 0, 0, 0, 0, 0)$ and we enter in the *while* circle until the iterations is greater than the maximum iteration parameter.
An action is generated randomly, let's suppose, it is E. The execution of this action moves the robot of about one unit in the east direction, see *Figure 2.2.*
The observation returned after the execution of this action by the robot is:

$$\mathbf{x}^{(2)} = (3.49, 0.49, 1.50, 0.49, 0.49, 3.49)$$

6. ALP calls *best_current_state*($\mathbf{x}^{(2)}$) function. The algorithm, in this case has to understand that is in a different state as before, using the perception function. In fact X_1, X_3,X_5 and X_6 will give us a low values returned by the factored perception function, i.e. under the threshold.

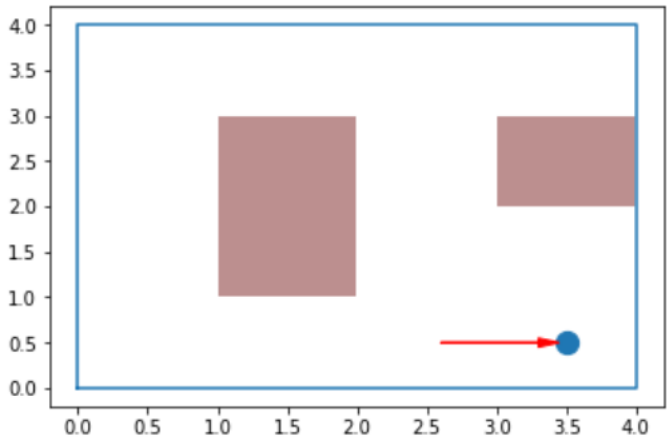

Figure 2.2: *In this configuration the robot executes the action E, represented with a red arrow. The position of the robot is symbolized by the blue dot in the coordinates (3.5,0.5). In this case the perception values that the robot return are (3.49,0.49,1.50,0.49,0.49,3.49).*

7. As consequence we need to extend the domains D_1, D_3, D_5 and D_6.

$$D_1 = \{0, 1\}, D_2 = \{0\}, ..., D_6 = \{0, 1\}$$

As a consequence we have the set of states S:

$$S = dom(\mathbf{V}) = \times_{i=1}^{6} D_i$$

ALP also extend the perception function for the new assignments as before.

8. Calculating the factored perception function for the all elements in the domains, the current state is set to $(1, 0, 1, 0, 1, 1)$. And all the observations are stored in the history of observations $\mathcal{O}$, and the transition

$$((0, 0, 0, 0, 0, 0), E, (1, 0, 1, 0, 1, 1))$$

is stored in the history of transitions $\mathcal{T}$.

9. Right now ALP call *update_f*() using the formula 2.14 and 2.15 calculated in the *Example 2.6*, and *update_gamma*() to update the transition function using the formula 2.16, so we extend the transition function with:

$$\gamma((0, 0, 0, 0, 0, 0), E) = (1, 0, 1, 0, 1, 1)$$

10. Finally we update the minimum likelihood using the parameter ϵ:

$$min\_likelihood = min\_likelihood \cdot (1 - \epsilon) \cdot e^{-\epsilon_d \cdot iter}$$

2.3 Learning the Transition Function

2.3.1 Problem Definition

Let us consider the computational aspects of learning a state-transition system $\mathcal{D}$. If the set of states S, and the set of actions A are small enough, it may be feasible to create a table, a matrix or a graph, that contains $\gamma(\mathbf{s}, a)$ and for every $\mathbf{s}$ and a, so that the outcome of each action can be retrieved directly from the table. In some cases is too large to specify every instance of $\gamma(\mathbf{s}, a)$ explicitly. Therefore, the next goal is to find a faster way to learn gamma. So far to learn this function the robot must physically do the action and observe the effects. But let's see this example where it shows us that each action has a specific effect on the variables that can be learned.

Example 2.7. *(Grid) Let us suppose to be in the configuration describe in the picture 2.3.*

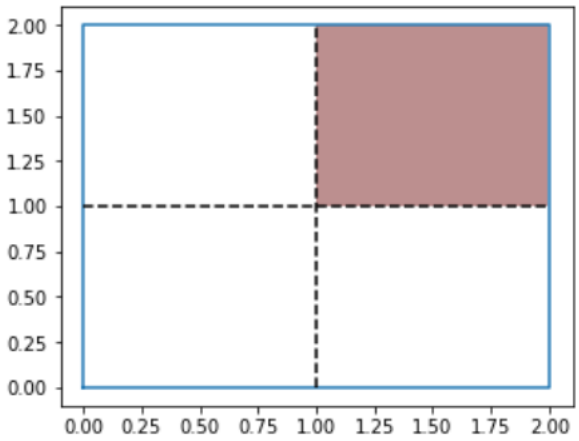

Figure 2.3: *In this configuration we are in a 2×2 grid with one closed cell.*

A graphical representation of the state transition system corresponding to the planning domain described in the Figure 2.3 is shown in Figure 2.4, where there is a graph the vertices correspond to the states and the arcs correspond to the transitions.

We can notice that each action has a specific effects on the variables, for example, the action North add more or less a unit to the x-coordinates, if there is no walls. In the case of walls there is no changes in the values of the variables except for some small sensing errors.

Let us formalize this concept. In this specific example we are working with Normal distribution. Therefore given an action a in A and a position, i.e. a

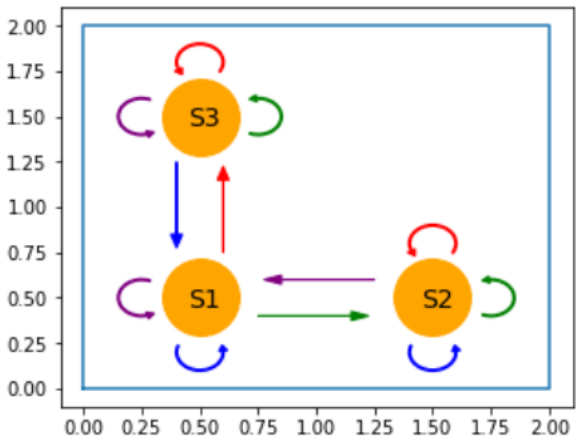

Figure 2.4: *A planning problem of a domain composed by 3 states:* $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \boldsymbol{s}^{(3)}$, *corresponding to 3 cells. No state for the closed cell. The states are symbolized by the big orange point. The actions are north, south, east and west and the transitions are represented by the colored arrows: red for the effect of action north, blue for south, green for east and finally purple for west.*

vectors of observations $\boldsymbol{x}$*=(x,y), then the conditional probability is:*

$$p_a^1(x'|x) = N(\mu_a^1(x), \sigma_a^1) \tag{2.22}$$

$$p_a^2(y'|y) = N(\mu_a^2(y), \sigma_a^2) \tag{2.23}$$

where μ_a *is some real function that maps* $\boldsymbol{x}$ *in the expected value after performing the action a. And* σ_a *is the model of the noise of the sensors associated to a. In particular*

$$\mu_{East}^1(x) = \begin{cases} x+1 & \textit{If there is no walls between x+1 and x} \\ x & \textit{otherwise} \end{cases}$$

$$\mu_{East}^2(y) = y$$

$$\mu_{West}^1(x) = \begin{cases} x-1 & \textit{If there is no walls between x-1 and x} \\ x & \textit{otherwise} \end{cases}$$

$$\mu_{West}^2(y) = y$$

$$\mu_{North}^1(x) = x$$

$$\mu_{North}^2(y) = \begin{cases} y+1 & \textit{If there is no walls between y+1 and y} \\ y & \textit{otherwise} \end{cases}$$

$$\mu^1_{South}(x) = x$$
$$\mu^2_{South}(y) = \begin{cases} y-1 & \textit{If there is no walls between y-1 and y} \\ y & \textit{otherwise} \end{cases}$$

This conditional probability expresses the probability of measuring $\boldsymbol{x}'$ *after executing the action a in a state in which the agent perceives* $\boldsymbol{x}$*. It represents the effects of execution the actions a in the real world.*

The usual approach is to develop a generative representation in which there are procedures for computing $\gamma(\mathbf{s}, a)$ given $\mathbf{s}$ and a.
For example, in [12], they assume that the transition function is specified with action language, resulting in a compact representation. They adopt an action languages which specifies the transition function through a set of rules of the form:

$$r : prec(r) \xrightarrow{a} eff(r)$$

where a is an action, $prec(r)$ is a propositional formula in the language of the constraints, and $eff(r)$ is a partial assignment.
But in this thesis we want to tackle the problem from another point of view.

Analyzing the previous example, we had see that each action has a specific effect on the variables, which can be learned. This suggests that we can use the information we have, to build a neural network that predicts to us the position of the robot given a state and an action.

Chapter 3

Predicting Actions Effects

In the first *Section 3.1* we briefly give an introduction of some concepts of artificial neural networks that we will use in the following sections.
In the second *Section 3.2*, we add to the Acting and Learning Planning domains algorithm a new part for predicting the effects of actions.
In the third *Section 3.3*, we implement the algorithm with the new part and we report some examples applied in our case studies. These examples help us to better explain the concept and the algorithm described in the previous section. And we also provide an analysis and an evaluation of the predictions.
Finally, in the last *Section 3.4*, we propose an idea to solve the problem of learning entirely the transition function that we will implement it in the *chapter 4*.

3.1 Brief Introduction to Artificial Neural Networks

An artificial neural network is a system consisting of interconnected units that compute non-linear function and they are inspired by human brain neurons. The definition of a neural network, is provided by the inventor of one of the first neurocomputers, Robert Hecht-Nielsen that defines a neural network as [14]:

> a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.

The basic computational unit of the brain is a neuron. A simple mathematical model for an artificial neuron is inspired by the biological neuron.
A neural network is a collection of artificial neurons connected together. Neurons are usually organized in layers. If there is more than one hidden layer the network is deep, otherwise it is called a shallow network.

If the network is a-cyclic, it is called a feed-forward network. Feed-forward networks are the commonest type of networks in practical applications. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes and to the output nodes. There are no cycles or loops in this network [14].
We can distinguish two types of feed-forward neural networks: single-layer perceptron see *Figure 3.1* and multi-layer perceptron see *Figure 3.2*. The most typical feed-forward network is a dense network where each neuron at layer $k-1$ is connected to each neuron at layer k.

Single Layer Perception

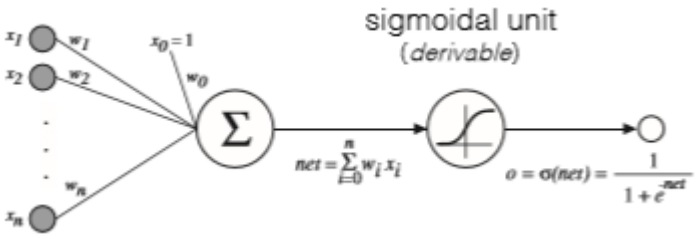

Figure 3.1: *Single perceptron with sigmoidal unit*

(i). **Input units**: represent the input variables and correspond to the impulses carried toward the neuron body in the human brain:

$$\mathbf{x} = (x_0, x_1, ..., x_n)$$

(ii). Adjustable **weights** are associated to connections among units:

$$\mathbf{w} = (w_0, w_1, ..., w_n)$$

(iii). **Output units**: represent the output variables that correspond to the impulses carried away from the cell body

$$out(\mathbf{x}) = \sigma(\sum_{i=0}^{n} w_i x_i) \tag{3.1}$$

(iv). **Activation function**: the activation function of a node defines the output of that node given an input or set of inputs. Some commonly used examples of activation functions are:

(i). Linear:

$$out(\mathbf{x}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

(ii). Sigmoidal:

$$out(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

(iii). Rectifield Linear ReLu:

$$out(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x}) = max\{0, \mathbf{w} \cdot \mathbf{x}\}$$

(iv). and many others.

Then we have the error function that measures the mean squared error of the target value with respect to the output.

$$E[\mathbf{w}] = \frac{1}{2N_{Tr}} \sum_{(\mathbf{x}^{(p)}, \mathbf{t}^{(p)}) \in Tr} (\mathbf{t}^{(p)} - out(\mathbf{x}^{(p)}))^2 \tag{3.2}$$

where N_{Tr} is the cardinality of the training set Tr.
Thus, $E[\mathbf{w}]$ should be minimized with respect to $\mathbf{w}$ using for example the stochastic gradient descent.

Multi layer perception

(i). d input units, $d + 1$ if the threshold is included into the weight vector:

$$\mathbf{x} = (x_1, ..., x_d)$$

(ii). N_H hidden units with output:

$$\mathbf{y} = (y_1, ..., y_{N_h})$$

(iii). c output units:

$$\mathbf{z} = (z_1, ..., z_c)$$

(iv). target vector:

$$\mathbf{t} = (t_1, ..., t_c)$$

(v). w_{ji} weight on the connection from input unit i to hidden unit j;

(vi). w_{kj} weight on the connection from hidden unit j to output unit k.

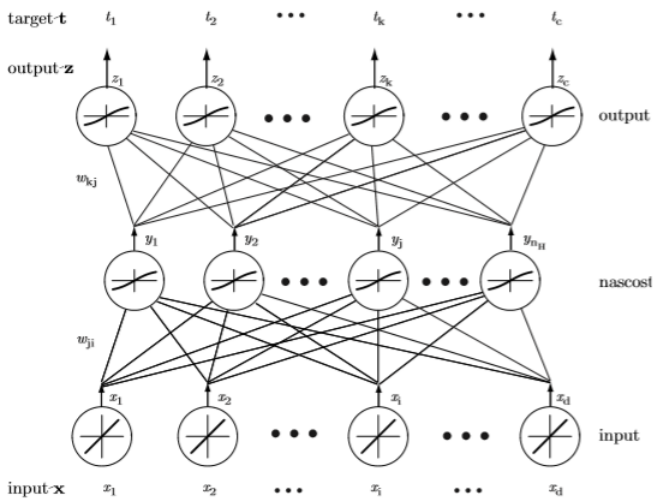

Figure 3.2: *Feed-forward neural network: Multi layer perception*

The network is defined by a matrix of parameters (weights) W_k for each layer. The matrix W_k has dimension $L_k \times L_{k+1}$ where L_k is the number of neurons at layer k. The weights W_k are the parameters of the model: they are learned during the training phase.
The number of layers and the number of neurons per layer are hyper-parameters: they are chosen by the user and fixed before training may start. Other important hyper-parameters govern training such as learning rate, batch-size, number of ephocs, and many others.

In the feed forward computation input units are set by some exterior function (it can be generated by sensors) which causes their output links to be activated at the specified level.
Working forward through the network, these outputs are going to be the input for the next layer. Each output is just the weighted sum of the activation on the links feeding into a node. The activation function transforms this linear combination: typically this is a non linear function (such a sigmoid) . This function correspond to the threshold of that node.

The function error, with c output units is:

$$E[\mathbf{w}] = \frac{1}{2cN_{Tr}} \sum_{(\mathbf{x}^{(p)},\mathbf{t}^{(p)})\in Tr} \sum_{k=1}^{c} (t_k^{(p)} - z_k(\mathbf{x}^{(p)}))^2 \tag{3.3}$$

where N_{Tr} is the cardinality of the training set Tr.
Thus, $E[\mathbf{w}]$ should be minimized with respect to $\mathbf{w}$ using for example the stochastic gradient descent.

3.2 Predicting Actions Effects

We can use this powerful structure of artificial neural network to predicts the perception variables given an action and starting perception variables.
We decide to use an online methods. So learning occurs during the execution phase. This kind of learning never stops, thus allowing the continuous correction and improvement of an incorrect model and adaptation to changing environment characteristics. On the flip side, the cost of learning is added to the cost of planning, augmenting the time window in which execution occurs.

Starting from ALP we collect the data that will constitute the training set to go to training the neural networks. These data are the perception variables of the transitions retrieved by the sensors. We suppose to have the modern sensors that allow us to collect large quantities of good quality data. So we can build accurate, non linear deep network models of these state transitions.

We model our state transition function as a deep neural network, one for each action. This deep neural network is a layered, a-cyclic, directed network structure. The number of layers and the number of neurons per layer are chosen by the user and fixed before the training start. Also the activation function, the loss function, and the optimizer can be choosen by the user and fixed at the starting point.
Specifically, in our cases we use as activation function a non linear (piece wise linear) Rectified Linear Units (ReLUs) of the form:

$$f(\mathbf{x}) = max\{\mathbf{x}, 0\}$$

In comparison to the other activation functions, such as sigmoid or hyperbolic tangent, ReLUs can be trained efficiently and permit direct compilation to a set of linear constraints.

Given a deep neural net configuration with L hidden layers, the following optimization problem has to be solved: $min\ E[\mathbf{w}]$.

The objective minimizes squared error of transitions in the data. And we use as optimizer the stochastic gradient descent.
We can see a pseudo-code of this procedure in the *Algorithm 3*.
In this algorithm we are going to add the predictive part to the structure of the ALP algorithm seen in *chapter 2*. As inputs other than the inputs explained in *section 2.1.1* we also have parameters for neural networks, such as the number of layers.
With the *predict_evaluation()* function we evaluate our forecast by comparing it with the true value of the sensing variables measured after the execution of the action. When this assessment is constantly below a predetermined threshold then it will mean that our neural network has learned the effects of actions and can be used to predict the effects of actions without having to do them physically.

input : initial planning domain: $\mathcal{D} = \langle S, A, \gamma \rangle$
input : initial perception function: f
input : perception initialization: p_{init,X_j}
input : parameters: α, β, ϵ
input : number of iterations: max_iter
input : empty initial neural networks

$\mathcal{O} \leftarrow \{\}$ # *Empty history of observations* ;
$\mathcal{T} \leftarrow \{\}$ # *Empty history of transitions* ;
$\mathcal{X} \leftarrow \{\}$;
$\mathcal{Y} \leftarrow \{\}$;
$\mathbf{x} \leftarrow$ sense() ;
current_state $\leftarrow$ best_current_state() ;
prediction $\leftarrow$ None ;
while *iter* $<$ *max_iter* **do**
  if *some conditions* **then**
    fit($\mathcal{X}$[a], $\mathcal{Y}$[a])
  end
  a $\leftarrow$select_action();
  prediction $\leftarrow$ predict() ;
  execute_action(a) ;
  $\mathbf{x} \leftarrow$ sense();
  current_state $\leftarrow$ best_current_state() ;
  $\mathcal{X} \leftarrow$ bring up-to-date ;
  $\mathcal{Y} \leftarrow$ bring up-to-date ;
  $\mathcal{O} \leftarrow$ bring up-to-date ;
  $\mathcal{T} \leftarrow$ bring up-to-date ;
  prediction_evaluation();
  update_gamma();
  update_f();
  iter +=1;
end

Algorithm 3: Pseudo-code for ALP with the Prediction Part

3.3 Case Study: Actions Predictions on Grid, and Grid with Walls Sensing

3.3.1 Examples of Actions Predictions

Example 3.1. *(Grid) In this example we suppose again to be in the situation described in the section 1.3.1. In this framework we are working with Normal distribution. Therefore given an action a in A and a position, i.e. a vectors of observations* $\boldsymbol{x}$*, then the conditional probability is:*

$$p_a^1(x^{'}|x) = N(\mu_a^1(x), \sigma_a^1) \tag{3.4}$$

$$p_a^2(y^{'}|y) = N(\mu_a^2(y), \sigma_a^2) \tag{3.5}$$

where μ *is some real function that maps* $\boldsymbol{x}$ *in the expected value after performing the action a. And* σ *is the model of the noise of the sensors associated to a. In particular*

$$\mu_{East}^1(x) = \begin{cases} x+1 & \textit{If there is no walls between x+1 and x} \\ x & \textit{otherwise} \end{cases}$$
$$\mu_{East}^2(y) = y$$

$$\mu_{West}^1(x) = \begin{cases} x-1 & \textit{If there is no walls between x-1 and x} \\ x & \textit{otherwise} \end{cases}$$
$$\mu_{West}^2(y) = y$$

$$\mu_{North}^1(x) = x$$
$$\mu_{North}^2(y) = \begin{cases} y+1 & \textit{If there is no walls between y+1 and y} \\ y & \textit{otherwise} \end{cases}$$

$$\mu_{South}^1(x) = x$$
$$\mu_{South}^2(y) = \begin{cases} y-1 & \textit{If there is no walls between y-1 and y} \\ y & \textit{otherwise} \end{cases}$$

In this simple example, this leads us to reflect on what role our neural network must play to learn the effect of actions in a given position. As we can see, we have a nonlinear function to learn.

$$\boldsymbol{x}_1 = W \cdot \boldsymbol{x} + \boldsymbol{b}$$

where $W = \begin{vmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{vmatrix}$ *and* $\boldsymbol{b} = \binom{b_1}{b_2}$.
As we have already seen in the Example 2.7 each actions have a specific effect on the variables.
In the ideal case of an infinite grid devoid of external walls, this simple network would represent the effect of an action.
For example in the case of north the matrix W *is the identity* $W = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$ *and the vector* $\boldsymbol{b}$ *is* $\boldsymbol{b} = (0, 1)$.
In the case of external walls we can use a deeper neural network or an activation function like

$$f(x) = \begin{cases} max\_values & x \geq max\_value \\ x & threshold \leq x < max\_value \\ slope(x - threshold) & x < threshold \end{cases}$$

Example 3.2. *(Grid and Walls Sensing) In this example we are in the simulation described in the section 1.3.2, but in this case we have a grid composed by 49 cells and six closed cells.*
we can when we go to compare the real position of the robot that the predicted

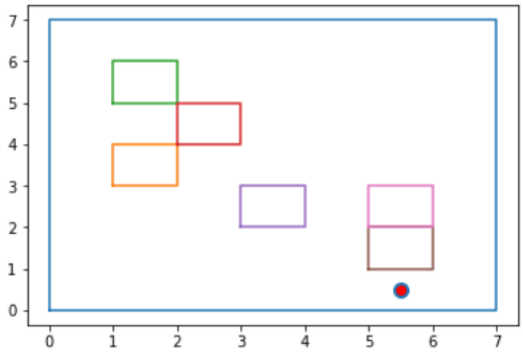

Figure 3.3: *In this configuration we have a* 7×7 *grid with six closed cell. In this figure we can see the real position of the robot marked with a blue dot and the predicted position with a red dot. We can argue that the two positions differ slightly and this is also due to the fact that we always have small measurement errors.*

position with our method. We note, in the figure, that the difference is really small. In fact, the euclidean distance between the actual position of the robot and the predicted position in this case is: 0.00224.
Now let's have the robot perform an action and compare it to the prediction. As an action we choose the North. We can see from the image 3.4 that a wall is present

in the north. So the robot will not crash but will remain in the starting position with some possible variance of sensing variables due to sensor measurement inaccuracies.

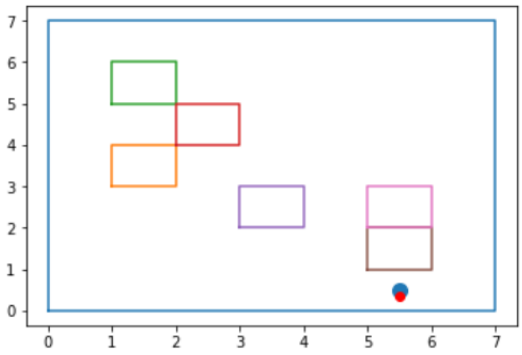

Figure 3.4: *In this configuration we have a 7×7 grid with six closed cell. In this figure we can see the real position of the robot marked with a blue dot and the predicted position with a red dot. We can argue that the two positions differ slightly and this is also due to the fact that we always have small measurement errors.*

In this case the euclidean distance between the actual position of the robot and the predicted position in this case is: 0.14921.

We can further make the robot perform an action, we do in the east, see figure

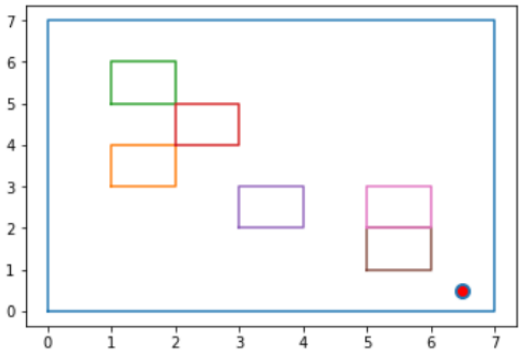

Figure 3.5: *In this configuration we have a 7×7 grid with six closed cell. In this figure we can see the real position of the robot marked with a blue dot and the predicted position with a red dot.*

3.6. Again we go to see the location of the prediction. In this case the euclidean distance between the actual position of the robot and the predicted position in this case is: 0.00149.

3.3.2 Evaluation of Predictions

Example 3.3. *(Comparison between Grid, and Grid with Walls Sensing) In this example we want to compare the quality of the predictions. In the situation of a* 7×7 *grid composed by 49 cells we want to analyze the difference between the prediction and the real effect of an action. We run the algorithm in the two simulators for 4000 iterations and we plot the mean of the euclidean distance in the Figure 3.7.*

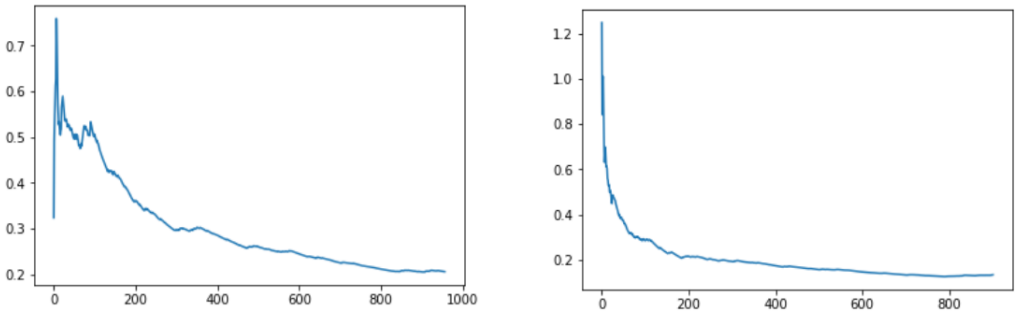

Figure 3.6: *(Grid) In these figure we run the ALP algorithm with the prediction part for 4000 iteration. In this plot we represent for the action East the mean of the euclidean distance between the real value after the execution of an action and the predicted values. The first figure in on the simulator Grid instead the second in the Grid with Walls Sensing. We can see that the line decreases during the iteration and goes around the 0.2. Obviously the prediction on the Grid with Walls Sensing is better and the line decreases quickly.*

We can see that in the case of the grid with wall sensing we have better results. In fact it goes faster under the 0.2. These better results are due to the better knowledge of the world, in fact we have also the distances from the walls that give us an improvement in the prediction of the neural network.

3.4 Predictions to Learning the Gamma Function

At this point we have a method that allows us to predict the effect of a given action given as input of the variables of perception. When this prediction has a low error value compared to the true value we can use it to predict the effects of our actions, without having to make the agent do it physically.
The idea is: the agent during its walk saves all the information observed. When it has enough information it can use this information to create neural networks that predict the effects of each action. When it realizes that this prediction is accurate, it can use it to complete the gamma function.

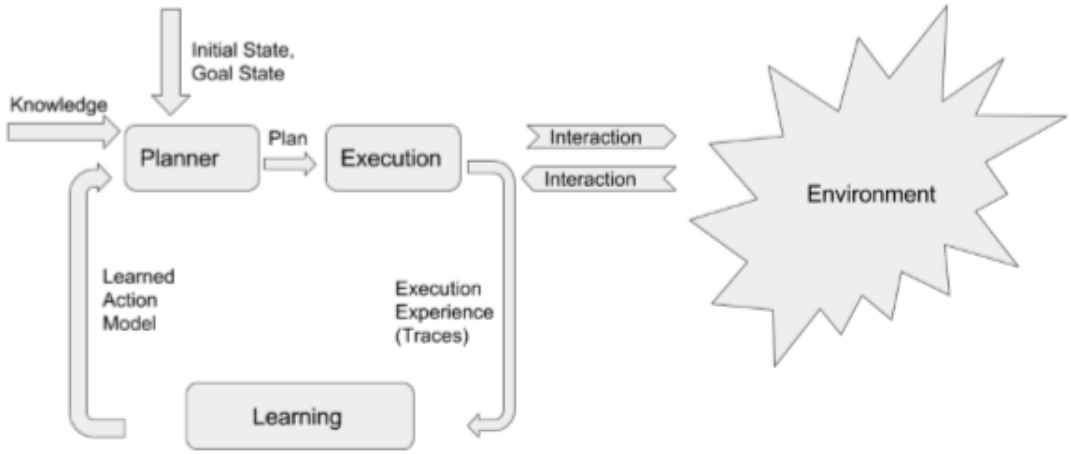

Figure 3.7: *In this figure we have a general scheme of the procedure, see [2]. We will see in detail in the chapter 4.*

In this figure we can see a general scheme of the procedure. As fundamental elements we have the planning and the execution of actions, which can have a specific goal state as purpose, we will see the definition of planning problem in the next chapter. And interaction with the external environment. From these experiences we obtain useful data to go to learn a learned action model, which precisely models the effect of actions.

Chapter 4

Completion of the Transition Function

In the first *Section 4.1* of this chapter, we dwell on the reasons why we want to learn the state-transition function. The ultimate goal of letting an agent learn a planning domain is to use it to built a strategy plan to achieve some goals. Therefore, in this first section we will define a planning problem, a plan, and the goal states, and we understand how we need the state-transition function.
In the next *Section 4.2*, we develop the ALP algorithm with the completion part of the transition function, in which the predictions are used to complete the state-transition function.
In the final *Section 4.3*, we implement the algorithm with the new parts, and we provide, as usual, an example in our simulators and results.

4.1 Motivations

The aims of the ALP algorithm is to learn the planning domain $\langle \mathcal{D}, f \rangle$, with $\mathcal{D} = \langle S, A, \gamma \rangle$, starting from an empty planning domains.
In particular we want that the algorithm learns the transition function γ, that assigns an arrival state in S, given a starting state $\mathbf{s} \in S$ and an action $a \in A$:

$$\begin{aligned} \gamma \colon S \times A &\longrightarrow S \\ (\mathbf{s}, a) &\longmapsto \gamma(\mathbf{s}, a) \end{aligned}$$

The new state $\gamma(\mathbf{s}, a)$ is the result of the action a in the state $\mathbf{s}$.
The knowledge of the entire gamma function can be useful to carry out a "plan strategy" to reach a goal. If we know the gamma function, given an initial state and a goal state, we can create a series of actions to achieve this.
In automated planning the agents decide how to achieve a goal given a set of

actions. These agents interact with the environment by executing actions which change the state of the environment, gradually propelling it from its initial state towards the agents' desired goal.
We begin with some definitions of fundamental concepts.

Definition 4.1. *(Planning Problem) A planning problem $\mathcal{P}$ is a triplet composed of:*

$$\mathcal{P} = \langle \mathcal{D}, \boldsymbol{s}_0, S_g \rangle \tag{4.1}$$

where:

- *the initial state $\boldsymbol{s}_0$ of the world;*
- *a set of goal states that is a subset of the set S:*

$$S_g \subset S$$

- *the planning domain $\mathcal{D}$ responsible for changing the world state by actions application.*

A solution to a planning problem in a deterministic domain is a plan that induces a sequence of states such that the last state is in the set of goal states.
The plan can be generated by applying some planning algorithm for deterministic domains. To do all this the knowledge of the entirely planning domain is needed, especially the transition function.

Definition 4.2. *(Policy) [3] A policy is a mapping between the world states and the preferred action to be executed in order to achieve a given set of goals.*

Definition 4.3. *(Plan as a policy) A plan π for $\mathcal{D}$ is a policy, i.e. a partial function from S to A.*

$$\begin{aligned} \pi \colon \overline{S} &\longrightarrow A \\ \boldsymbol{s} &\longmapsto \pi(\boldsymbol{s}) \end{aligned}$$

Thus, starting in the initial state s_0 ,following the plan π, the next action is:

$$\pi(\mathbf{s}_0) = a_1$$

Therefore, in deterministic domains, when a initial state $\mathbf{s}_0$ is given, a plan can be see as a sequence of actions that, when performed from an initial state induces a sequence of states.

$$\gamma(\mathbf{s}_0, a_1) = \mathbf{s}_1$$

Definition 4.4. *(Plan as a sequence of actions [2]) A plan, given an initial state of the system, a goal, and a planning domain, is a sequence of actions* $\pi = [a_1, a_2, ..., a_n]$ *that drives the system from the initial state to the goal.*

The transition from the initial state to the goal is driven by the previously mentioned transition function.

$$\gamma(\mathbf{s}, \pi) = \begin{cases} \mathbf{s} & if \quad |\pi| = 0 \\ \gamma(\gamma(\mathbf{s}, [a_2, ..., a_k]) & otherwise \end{cases}$$

This transition function, when applied to the set of the current state and the action, produces the next state as:

$$\mathbf{s}_{i+1} = \gamma(\mathbf{s}_i, a)$$

This transition function, when successively applied to the resulting intermediate states at each step, leads to the goal. Each such action sequence, complete with initial state and goal information (and occasionally intermediate state information), constitutes a trace. These traces can either be in the form of action sequences or action sequences interleaved with intermediate states [2].

But, we have see in our algorithm that it has some limitations to learning the entirely gamma. In fact, to learn the transition function entirely, which describes the effects of actions on the environment, the agent has to physically do and perform the actions and then, from the perceptions, it learns the transitions.
This can become prohibitive in the case of a large number of states and actions, so we need an huge number of iterations to learn entirely the transition function. This is why we will use the predictions described in chapter 3, to complete the transition function.

4.2 Complete the Transition Function with Predictions

We want to define a function that have the following objectives:

- use the predictions to complete the transition function as much as possible;
- if during the walk we improve the forecasts, update the previous completion of the gamma function.

The general schema is:

Loop for all states and actions:

1. *select a state;*
2. *select an action;*
3. *check if we have already do this transition;*
4. *if not, predict the next observation* $\boldsymbol{x}$*;*
5. *decide in which state it is, based on maximum likelihood. If the likelihood is too low for all the existing states, create a new state;*
6. *complete with this transition the gamma function;*

We will see in details in the next sections all these steps.

4.2.1 Description of the Algorithm

To deal with this problem to complete the transition function we decided to use not only the physical observations but also the predictions of our neural network.

When we have a good predictions, i.e. the euclidean distances in under a fixed threshold, then we can use these predictions to complete the transition function. In simple terms, this means that when the agent is sure that its neural network is good and given the perceptions variables that correspond to a state $\mathbf{s}$ in S, and an action a in A, correctly predicts the arrival perception variables. Then this can be suitable to help us to complete the gamma function.
It is as if the agent just realizes that it has a good neural network to predict actions, it stops and starts thinking.
In its reflection the agent thinks about all possible states and possible actions and tries to predict the various transitions. This is done for all states in S and for all actions in A that have not already been physically performed.
It the algorithm 4, we have a pseudo-code for a function which will be inserted in the ALP algorithm, called *complete_gamma_function()*. This function complete and update the transition function using the predictions.

As input there are the planning domain $\langle \mathcal{D}, f \rangle$ learned till now,with an incomplete transition function, and the history of transitions $\mathcal{T}$.

With a for loop on states and one for the actions, for each pair $(state, action)$ we have to check if we have already performed this action in this state.
To do this, we go to check, in the dictionary of the history of the transition $\mathcal{T}$, if this action has already been performed physically in this state.

```
input  : planning domain: ⟨D, f⟩ with D = ⟨S, A, γ⟩
input  : history of transition T
for s in S do
    for a in A do
        if (s,a) in T then
            break
        end
        prediction ← predict() ;
        predicted_state ← best_state() ;
        add_to_gamma();
    end
end
```

Algorithm 4: Pseudo code for *complete_ gamma_function()*

If we are in the case that we have already performed this couple, it makes no sense to make a prediction, since we already have the real true observations.
In the event that this $(state, action)$ pair has never been performed, then we proceed with the prediction.
We use our neural network to predict the output variables, calling the function *predict()*. As input we give as perceptions variables the values gives to us using the perception function related to these state variables. Therefore we use this predicted perception variables to find the correct state, calling the function *best_state*(). This function selects the best state given these perceptions, for detail see Observation 4.6.
And finally, we update the transition function with the new triple:
$(state, action, predicted\_state)$.

4.2.2 Observations

Observation 4.5. *The first observation concerns the first for loop. We said that we want to complete the transition function for all states in our domain. But this depends on the definition of states.*
Recall that in chapter 1 we defined the set of states as a subset of the cartesian product of the domains.

$$S \subseteq dom(\boldsymbol{V})$$

And that it does not necessarily coincide with the set of observed states. So a choice would be to complete the transition function only in the really observed states. We will see in detail with the example 4.9 the differences of the two possible choices.

Observation 4.6. *Another observation concerns the best_state() function, which*

given the prediction of the variables of perception, goes to find the state that maximizes likelihood. In this case, the agent can decide whether to be impulsive and believe in its predictions, or to behave more cautiously. In the first case, it firmly believes in its predictions and if it predicts a perception that is below threshold for each state, it allows the extension of the domains. In the second case it is more cautious and updates gamma only if it finds the perception among its states.

Observation 4.7. *These steps to learn the completion of the transition function can be done several times with a fixed frequency. Let's take an example, if after 1000 iterations the agent realizes that it has a good neural network that predicts with a low error the variables given a state and an action. Then the agent decides to complete the transition function as described before. After completing it, it continues its walk, acquiring other knowledge and observations. This new knowledge will improve the prediction accuracy of our neural networks. So it is quite natural to proceed again with another completion of the transition function, which in this case is an improvement on the previous predictions. Obviously only the couples (state, action) in which it has not yet physically gone, but had only previously been predicted, will be predicted.*

4.3 Case Study: Complete the Gamma Function on Grid with Walls Sensing

Example 4.8. *(Grid with Walls Sensing) We see some observations in a toy example. In this example we have few states and few actions for reasons of comprehensibility of the example. In this simple case it would be more convenient to physically visit the states. But it can be useful to see some important aspects.*
We start from a configuration described in the figure 4.1. We are in the case of the 'Grid with walls sensing' simulator described in the section 1.3.2.

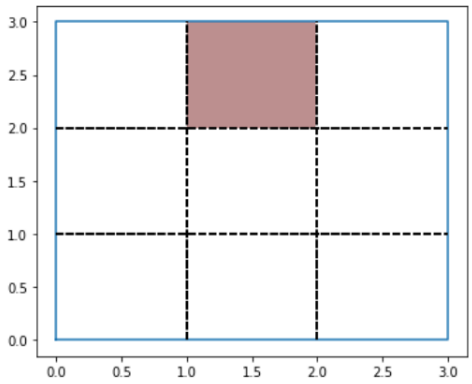

Figure 4.1: *In this configuration we have a 3×3 grid with one closed cell*

Suppose that after 50 iterations the planning domain learned, and especially the transitions in the transition function, are that shown in the figure in Figure 4.2.

We can see that we have only partial knowledge of the domain and transition function. Suppose in this simplified example that the algorithm predicts with a good accuracy the sensing variables given an action and the starting perception variables.

Learning starting from the observed state

So we apply the completion of the transition function on all observed states: $\{s1, s2, s3, s4\}$. *We can see that the transition learned are:*

- $\gamma(s3, W) = s3$ *this transition is correctly learned as it has both the data related to the previous transitions of the west action in states s1 and s4, and it has data related to the perception variables of the distance from the walls;*
- $\gamma(s3, S) = s1$ *this transition is correctly learned and the arrival state is between the observed states;*

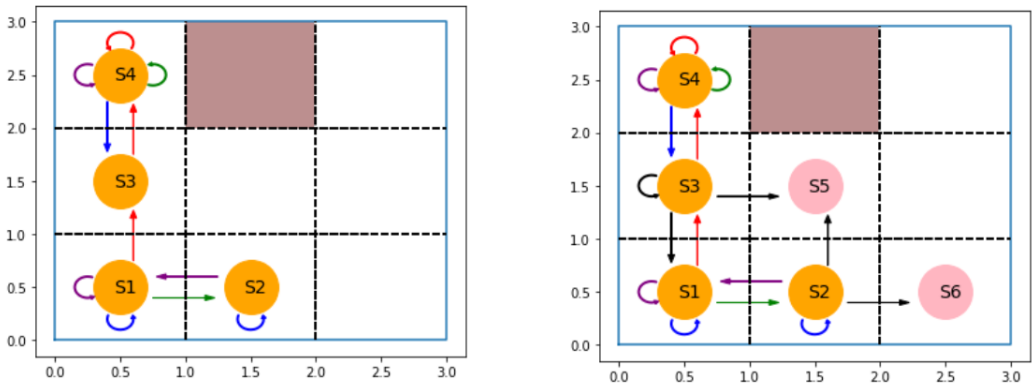

Figure 4.2: *In the first figure is represented the knowledge of the domain after 50 iterations. The states observed are marked with orange circles, and the colored arrows (one color for each action) indicate the transitions performed, and therefore present in the transition function. In the second figure we have in addition to the previous one the black arrows that indicate the transition learned using the predictions. With the pink dot we have states that are not observed but that we expect to exist and that are the predicted effects of action on the observed states.*

- $\gamma(s3, E) = s5$ *and* $\gamma(s2, N) = s5$ *these two transitions go into a state never observed but present in our set of states as a Cartesian product of the domains;*

- $\gamma(s2, E) = s6$ *in the latter transition, however, we have the forecast of a state not present in the state of states. So some variables are added to the domains.*

Learning starting from all the cartesian product of domain

In the case in which the set of states corresponds to the cartesian product of the domains we have that inside there have been both observed and hypothetical states. Among the hypothetical states we can have states that really correspond to a real state, and states that are instead fictitious, so they don't make sense. In this sense we can go to eliminate the latter. Obviously, if after a large number of iterations the robot realizes that it cannot reach a state in any way, then this can be used as evidence for the non-reachability of this state.

In this example we can see that the state s5 and s6 are inside S but have never been observed. We have a difference though. From state s4 we tried to go east without result and furthermore the distance of the walls measured in s2 give us evidence that it is not possible to reach it. And so also for other states in the cartesian product of domains.

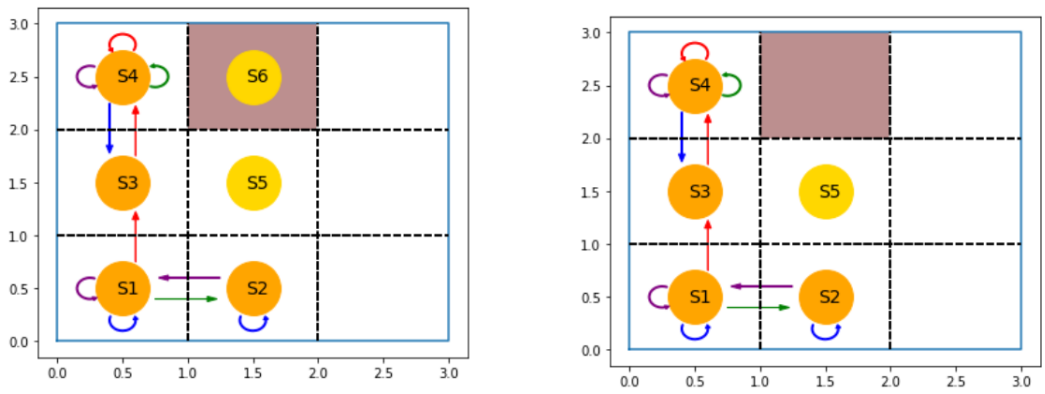

Figure 4.3: *In this figure we add respect to the previous ones some hypothetical states with the yellow dot.*

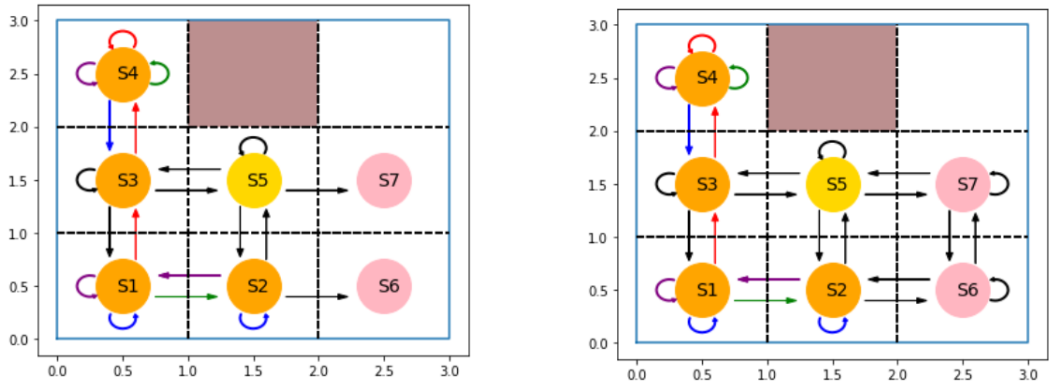

Figure 4.4: *Here we plot the transitions learned using the predictions.*

Therefore we can go to apply the completion of gamma not only on the observed states but also on other states that we hypothesize to be true. Obviously this will allow us to further complete the transition function, see Figure 4.4.

Chapter 5

Experimental Evaluation

In order to estimate the quality of the model generated by the ALP algorithm we should define some metrics, or methods, to measure the important aspects of our model, and compared it with the reality.
In particular, in the first *Section 5.1*, we evaluate the learned state space. We compare the number of learned states in the final planning domain learned by the algorithm, with the number of real states. In fact, we do not have a fixed number of states defined a priori, but we have a dynamic set that is learned during the running of the algorithm. And between them, we analyse how is the number of correct learned states that correspond to a real world state. Also we test for the completeness of the learned model. We provide examples and experiments that show this analysis, and we also test how the setting of the parameters influences the learning process, and the convergence to a model coherent with the world.
Finally, in the *Section 5.2*, we evaluate the transition function. We define a measure of coherence between the abstract model, i.e.the learned planning domain, and the real world as perceived by the agent. We provide examples and experiments that show this analysis.

5.1 Evaluating the State Space

5.1.1 Number of States

Given the extended planning domain of the agent $\langle \mathcal{D}, f \rangle$, with $\mathcal{D} = \langle S, A, \gamma \rangle$ and f is a perception function, a good model is when we have coherence between the states in abstract model and the real world states.
In the first experiment we want to compare the number of states in the real environment which we want to learn, and the number of states learned by the *Acting and Learning Planning domains* algorithm in the final model.

Let us denote with:

1. S the set of learned states returned by the algorithm in the final model, and respectively $|S|$ the number of these states founded;
2. $\widehat{S}$ the set of real desirable states of the real world and respectively we denote with $|\widehat{S}|$ the number of these states.

Experiment 5.1 (Grid)**.** *In this example we are in the framework described in the section 1.3.1, with this configuration: a 7×7 grid composed by 49 cells and we have one closed cell in the coordinate (2.5,2.5), see Figure 3.6.*

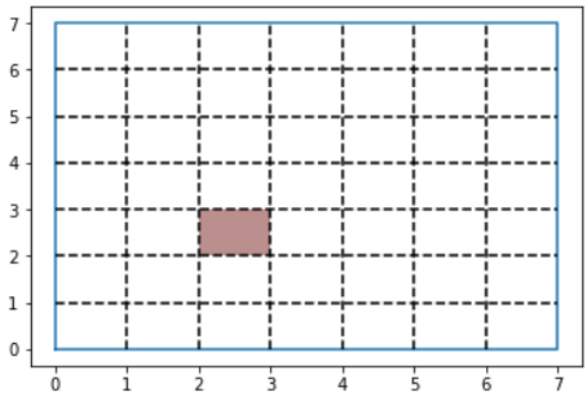

Figure 5.1: *In this configuration we have a 7×7 grid with only one closed cell. In the real world we expect to learn with the ALP algorithm a state for each cell. Except for the brown cell that correspond to a closed cell. Thus, in this case, we expect that the algorithm learns 48 states.*

In general, in this configuration, for a grid $n \times m$, the number of states that we want to learn are equal to the product $n \cdot m$ minus the closed cells $|C_{closed}|$.

$$|\widehat{S}| = n \cdot m - |C_{closed}|$$

Here, in this experiment, the number of real states are 48 because we have only one closed cells..

$$|\widehat{S}| = 7 \cdot 7 - 1 = 48$$

We suppose a one to one correspondence between the set of state variables and the set of domains, thus we have that $dom(V_1) = D_1$ and $dom(V_2) = D_2$. And the desirable domains that we want to learn are are:

$$D_1 = \{0, 1, 2, 3, 4, 5, 6\}$$
$$D_2 = \{0, 1, 2, 3, 4, 5, 6\}$$

If we run the algorithm code, with this configuration and ϵ=0.9, the number of states $|S|$ that are found after 1000 iterations are 56 .
We can see, in the Figure 3.7, how the number of learned states increase during the iterations. We can see that in the first 200 iterations the number increases and then after it become constant in 56.

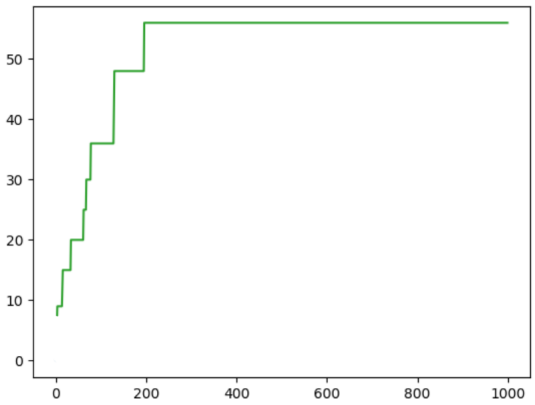

Figure 5.2: *In this plot in the x-axis there are the iterations from 0 to 1000 and in the y-axis there are the numbers of states.*
We can notice that the number of states increases with the number of iterations and then after more or less the iteration 200 the number become constant in 56.

We run the algorithm 10 times and then we average the number of learned states of the final model. If we run 10 times the algorithm in this configuration we found the following results for $|S|$:

$$\{56, 70, 63, 49, 64, 56, 72, 56, 98, 63\}$$

For all this experiments it can be notice there is an incremental part where the number of states increasing during the iterations. And a second part of the plot where we can see that the number become constant, so the algorithm has finished to learn states and converges.
We can see, in the Figure 5.3 , the plot of the average of the number of the states in 20 iterations, and we can confirm the behaviour described before.

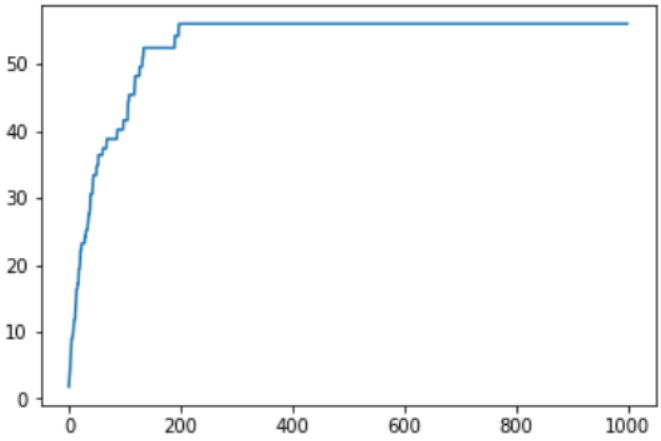

Figure 5.3: *In this graph in the x-coordinates there are the iterations from 0 to 1000 and in the y-axis there are the mean of the numbers of states. In this case we plot the mean in 20 iterations. We can confirm that the number of states increases with the number of iterations and then the number become constant in 59.57.*

5.1.2 Observed States and Hypothetical States

Between the learned states in S, an important issue is the distinction and correspondence between hypothetical states and observed states, see [5]. When an agent reasons about what might happen and simulates changes of state to assess how desirable a course of action is, it uses predicted states, as we have seen in Example 4.8. Predicted states are in general less detailed than the observed one; they are obtained as a result of predictions that may be not correspond to the reality.

We have seen how many states the algorithm learns, but now we want to analyse how many of these state in S are actually correct and correspond to a real state in the world. Let us do a simple example to clarify further.

Example 5.2. *(Grid) Let us now suppose to be in this situation, a* 2×2 *grid composed by 4 cells. Let us suppose that the robot during its walk, it has visiting three states, see Figure 5.4 .*

During the walk the robot updates his domains in:

$$D_1 = \{0, 1\}$$
$$D_2 = \{0, 1\}$$

Thus, we supposing that the observed states are (0,0),(0,1) and (1,0). At this stage the agent can conjecture that there is another cell in $(1, 1)$*. In fact we have that:*

1. $S = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ *and* $|S| = 4$ *;*

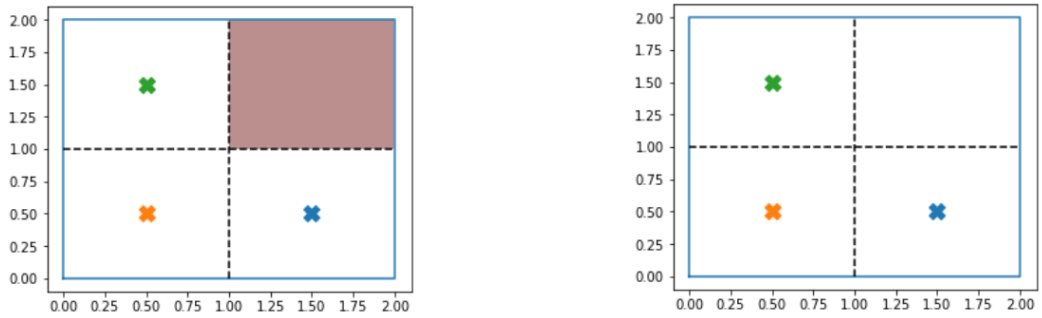

Figure 5.4: *In the both configurations represented in this picture we have a* 2×2 *grid with 4 cells. The robot during his walk has visited 3 states represented by the colored cross. But in the first situation we have that the remain undiscovered cell is closed, instead in the second configuration we have an open cell but it is not observed yet during the walk of the robot. How the robot can distinguish between these two situations given his observations?*

2. $\widehat{S} = \{(0,0),(0,1),(1,0)\}$ *and* $|\widehat{S}| = 3$.

Therefore the agent speculates that exist another possible state in (1,1) but it never been in it for now.

But how can it distinguish between the two situations represented in the figure 5.4? There is the possibility to reach the state (1,1) or not?

We can divide the learned states in S on:

- S_{osb} that are states in which the robot has been physically.
- S_{hyp} that are states that are hypothesised.

Obviously, if after a large number of iterations the robot realizes that it cannot reach a state in any way, then this can be used as evidence for the non-reachability of this state.

Experiment 5.3. *(Grid) In this example we are in the same configuration described in the Figure 5.1. We want to analyse better these 56 states founded. We can see that after 200 iterations the robot has not visited yet physically all the possible states in the real world, but it can speculate the other states. Looking at the figure 5.5, we can notice that the robot has been in only 39 states physically. The other states are only hypothesised. We can also notice that among the hypothesised states there are some that correspond to a real states, but also one in the closed cell.*

We can plot the same results after 1000 iterations and we can see that the robot has been in all the real states. And we can realise that we never been in the closed cell. This can be seen as an evidence for the non reachability of this states.

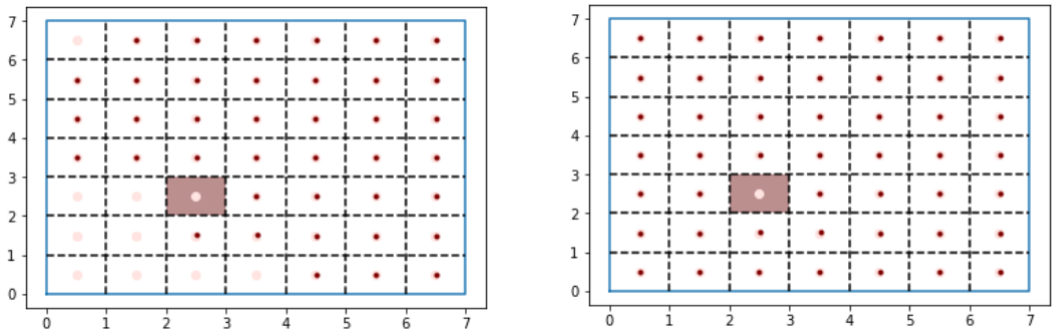

Figure 5.5: *In this case we are in the configuration of a 7 times7 grid with one closed cell. In the first figure we plot in the grid with a pink point the states hypothesised after 200 iterations and with a red point the states observed, in which the robot has been physically. We can notice that the robot has been in only 39 states physically. The other states are only hypothesised. We can also notice that among the hypothesised states there are some that correspond to a real states, but also one in the closed cell. In the second figure we plot in the grid with a pink point the states hypothesised after 1000 iterations and with a red point the states observed, in which the robot has been physically. We can notice that the robot has been in all the real states physically. We can also notice that the closed states remains only hypothesised.*

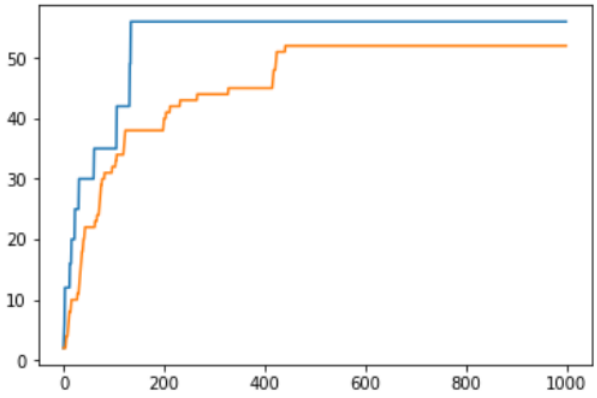

Figure 5.6: *In this plot in the x-axis there are the iterations from 0 to 1000 and in the y-axis there are the numbers of states. With the blue line we can see the number of state learned. We can notice that the number of states increases with the number of iterations and then after more or less the iteration 200 the number become constant in 56. Instead with the orange line we can see the number of state observed during the iterations.*

5.1.3 Redundant States

In the next experiment we want to see among these learned states how many effectively are correct and in other hand, how many do not correspond to a real

state of the world.

Experiment 5.4. *(Grid) In this example we are in the configuration described before in the Experiment 3.1. We want to analyse better these 56 states founded.*

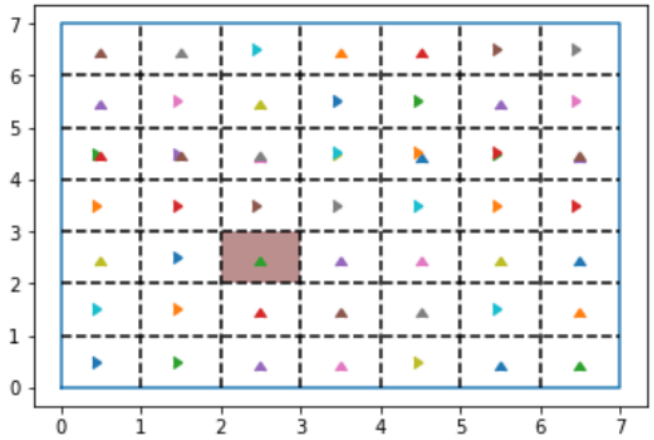

Figure 5.7: *In this we plot in the grid the mean of the 56 states learned by the algorithm after 1000 iterations and with the parameter ϵ set to 0.9. We can notice that we have a state that does not exist because of the wall in the coordinates (2.5,2.5). The other states are correct but in same cases we have learned two state for one real state, caused by some small measurement errors or instruments sensitivity of the sensors.*

We can notice that there is a state in the closed cell. This state does not correspond to any real state because of the wall. The other states are correct but in same cases we have learned two state for one real state, caused by some small measurement errors or instruments sensitivity of the sensors.

5.1.4 Completeness

We also want to prove the completeness of our learned domain. Completeness in our case, means that given all the possible positions that our agent can have in the real world, i.e. all the possible values of my perceptions variables, we have a state that represents this perception in my model. That is, a state that given the observed vector is above likelihood.

Experiment 5.5. *(Grid) In this example we are in the same configuration described in the Figure 3.6. In order to estimate the completeness we run the robot and given the various positions we test whether or not there are states that represent it.*

Table 5.1: In this table we have in the first column the number of iterations. In the second column we have the completeness in percentage of 10000 possible positions.

number of iterations	percentage of completeness
100	0.946
500	0.998
1000	1.0

5.1.5 Effects of the Configuration of the Epsilon Parameter

Effects of the parameter ϵ in the number of states

In the ALP algorithm we have as input some parameters. These parameters allow to the agent to define its behaviour, if it wants to have an approach more cautious or the other way around, more impulsive. And they express in some way how much the agent trusts in the structure of the model. In fact, each of these parameters represent a key component of the model, and particularly ϵ for the states. We want to analyse how the choice of the values of the ϵ influence the number of learned states $|S|$.

Experiment 5.6. *(Grid) In this example we are in the same configuration as before, described in the Figure 5.1 .*

Let us now consider the effects of the parameter ϵ, the experiments result are summarized in Table 5.2. Where we compare the number of learned states changing the values of parameter $\epsilon \in \{0.0, 0.5, 0.9, 1.0\}$. The results in the $|S|$ column are the average on 10 runs.

Table 5.2: In this table we have in the first column the values of parameter ϵ that are $\{0.0, 0.5, 0.9, 1.0\}$. In the second column we have the average of the new learned states in ten runs of the algorithm in the configuration shown in Figure 5.1 with 1000 iterations. In the third column we compare the average number of the new learned states with the right one values that in this case is 48. With this table we can compare the situation changing the values of parameter ϵ.

value of parameter ϵ	nr of learned states: $\|S\|$ (average on 10 runs)	nr of desirable states: $\mathcal{S}$
$epsilon = 0.0$	348,5	48
$epsilon = 0.5$	184,3	48
$epsilon = 0.9$	54,7	48
$epsilon = 1.0$	1	48

We can notice that:

(i). $\epsilon = 1$ *: in all cases no new state is created, only the first initial state. In this case we do not learn nothing.*

(ii). $\epsilon = 0$ *: a lot of states are created, too many for represent our world.*

(iii). $\epsilon \in (0, 1)$*: we are in the middle, especially in case of epsilon* $= 0.9$ *the number of new learned states is slightly larger than the right one.*

5.2 Evaluating the Transition Function

5.2.1 Coherence

Given the extended planning domain of the agent $\langle \mathcal{D}, f \rangle$, with $\mathcal{D} = \langle S, A, \gamma \rangle$ and f is a perception function, a good model is when we have coherence between the abstract model and the real world effects of an action perceived through the perceptions variables.
Intuitively, if

$$\gamma(\mathbf{s}^{(1)}, a) = \mathbf{s}^{(2)}$$

then if the agent perceives to be in $\mathbf{s}^{(1)}$ and it performs a, then after the execution of a it will perceive to be in the state $\mathbf{s}^{(2)}$.
But if $\mathbf{s}$ is the state that maximises the product of perception function and we have that it is different from the state predicted by the transition function of our planning domain:

$$\mathbf{s} \neq \gamma(\mathbf{s}^{(1)}, a) = \mathbf{s}^{(2)}$$

this means that it perceives to be in a different state respect the prediction of the state-transition function, then we have an incoherence between the real world and our model.

Experiment 5.7. *(Grid) In this example we are in the same configuration described in the Figure 3.6. In order to estimate the quality of the learned transition function we compare the learned abstract model and the real world effect. For each state and for each action we compare the state that maximised the likelihood and the state modelled by the domain in the transition function. The results can be seen it the table 5.3.*

Table 5.3: In this table we have in the first column the number of iterations. In the second column we have the coherence in percentage of the coherent states and in the last column the percentage of the incoherence.

number of iterations	percentage of coherence	percentage of incoherence
500	0.8125	0.1875
1000	0.8625	0.1375
2000	0.8973	0.1027

Conclusions

In this master thesis we have used a formal framework for the online construction of an abstract planning domain. This framework provides the ability to learn planning domains where the states are represented with state variables which are described by a set of variables taking values over domains. The aim of the thesis is to use automatic methods to learn these models by carrying out actions, and observing their effects on the environment.

Learning a finite deterministic planning domain represented with state variables opens up the possibility to use all the available efficient planners to reason at the abstract level. Learning the perception function takes into account the fact that, while an agent can conveniently plan at the abstract level, it perceives the world and acts through sensors and actuators that work in a continuous space. Learning perception functions allows us to learn new states that represent unexpected situations of the world. This framework also allows us to learn domains incrementally, and to adapt to a changing environment.

We have deal with the problem of learning entirely the state-transition function. Extending the ALP algorithm with a prediction part permit us to predict the effects of the actions in a state without doing physically the actions and perceives the sensor variables. These predictions are useful to complete the state-transition function that can be useful for efficient planners to solve a planning problem.
Our experiments show the convergence to coherent model and validate our analysis.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Arora, A., Fiorino, H., Pellier, D., Metivier, M., and Pesty, S. A review of learning planning action models. *The Knowledge Engineering Review 33* (2018).

[3] Celorrio, S. J., and de la Rosa Turbides, T. Learning-based planning. In *Encyclopedia of Artificial Intelligence*. IGI Global, 2009, pp. 1024–1028.

[4] Cimatti, A., Pistore, M., and Traverso, P. Automated planning. *Foundations of Artificial Intelligence 3* (2008), 841–867.

[5] Ghallab, M., Nau, D., and Traverso, P. *Automated Planning: theory and practice*. Elsevier, 2004.

[6] Gregory, P., and Cresswell, S. Domain model acquisition in the presence of static relations in the lop system. In *ICAPS* (2015).

[7] Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., and Davidson, J. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551* (2018).

[8] Henaff, M., Whitney, W. F., and LeCun, Y. Model-based planning in discrete action spaces. *arXiv preprint arXiv:1705.07177* (2017).

[9] Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research 29* (2007), 309–352.

[10] Say, B., Wu, G., Zhou, Y. Q., and Sanner, S. Nonlinear hybrid planning with deep net learned transition models and mixed-integer linear programming. In *IJCAI* (2017), pp. 750–756.

[11] Serafini, L., and Traverso, P. Incremental learning abstract discrete planning domains and mappings to continuous perceptions. *arXiv preprint arXiv:1810.07096* (2018).

[12] Serafini, L., and Traverso, P. Incremental learning of discrete planning domains from continuous perceptions. *arXiv preprint arXiv:1903.05937* (2019).

[13] Serafini, L., and Traverso, P. Learning abstract planning domains and mappings to real world perceptions. In *International Conference of the Italian Association for Artificial Intelligence* (2019), Springer, pp. 461–476.

[14] Shanmuganathan, S. Artificial neural network modelling: An introduction. In *Artificial neural network modelling*. Springer, 2016, pp. 1–14.

[15] Zhuo, H. H., and Yang, Q. Action-model acquisition for planning via transfer learning. *Artificial intelligence 212* (2014), 80–103.