Sapienza
Università di Roma

Reinforcement Learning for $\text{LTL}_f/\text{LDL}_f$ Goals: Theory and Implementation

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Marco Favorito
ID number 1609890

Thesis Advisor

Prof. Giuseppe De Giacomo

Academic Year 2017/2018

Reinforcement Learning for LTL$_f$/LDL$_f$ Goals: Theory and Implementation
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: favorito.1609890@studenti.uniroma1.it

Abstract

MDPs extended with $\text{LTL}_f/\text{LDL}_f$ non-Markovian rewards have recently attracted interest as a way to specify rewards declaratively. In this thesis, we discuss how a reinforcement learning agent can learn policies fulfilling $\text{LTL}_f/\text{LDL}_f$ goals. In particular we focus on the case where we have two separate representations of the world: one for the agent, using the (predefined, possibly low-level) features available to it, and one for the goal, expressed in terms of high-level (human-understandable) fluents. We formally define the problem and show how it can be solved. Moreover, we provide experimental evidence that keeping the RL agent feature space separated from the goal's can work in practice, showing interesting cases where the agent can indeed learn a policy that fulfills the $\text{LTL}_f/\text{LDL}_f$ goal using only its features (augmented with additional memory).

Contents

Chapter 1

Introduction

This chapter presents the outline of this thesis and summarizes motivations, goals, and achievements. In Section 1.1 we generally describe the field of Reinforcement Learning, by explaining what it is, the Markov Decision Process models (MDP) and the main challenges that arise in solving them. In Section 1.2 we focus on a specific subset of Reinforcement Learning problem, known in literature as Non-Markovian Reward Decision Process (NMRDP), that defines the background for our work. We explicitly declare our goals and the reached achievements in Section 1.3 and Section 1.4, respectively. Finally, the chapter ends with Section 1.5 by summarizing the structure of the thesis.

1.1 Reinforcement Learning

Reinforcement learning is an area of Machine Learning where the learning comes from rewards and punishments (Sutton and Barto, 1998). It is concerned in how the learning entity, the *agent*, interacting in an *environment*, should take *actions* so to maximize the observed *reward*. The reward signal is observed after each action taken by the agent. The agent chooses actions depending on the current state of the environment. A solution to the reinforcement learning problem is a *policy* which determines which action should be executed in a given state in order to maximize the long-term reward. An algorithm that tackles this kind of problem is called *reinforcement learning algorithm*.

The problem, due to its generality, is studied in many other disciplines, such as *game theory*, *control theory*, *operations research*, *multi-agent system*, and many others. Usually, in order to simplify the tractability of the problem, it is assumed that the environment can be modelled as a *Markov Decision Process* (MDP). An environment behaves like an MDP if the Markov property is satisfied, which means that the state space representation in the algorithm captures enough details so that the optimal decisions can be made when the information about only the current state is available.

Even if the laws that determine the evolution of the systems and the rewards are unknown a priori, it is still possible to solve an MDP by making several simulations and gathering experiences about the visited states, the actions taken and the observed rewards. However, many challenges arise in this settings:

- *Exponential state space explosion*: due to the feature selection used for the state space encoding, every added feature yields an exponential increase in the number of states, hence reducing the tractability of the problem.

- *Exploration-Exploitation trade-off*: due to the former issue, reinforcement learning algorithm should be designed to avoid exploring irrelevant states in terms of expected reward, while preferring the ones with high expected reward (*exploitation*). Furthermore, the algorithms should be sensitive to the local optima issue, a well-known in statistical learning literature (*exploration*).
- *Temporal credit assignment*: the agent should be able to foresee the effect of his actions (in terms of expected reward), due to the fact that, in many domains, the current reward is influenced by past decisions.

These problems lead to the use of heuristics and approximate solutions. A simple way to do reinforcement learning is to use exploration which is based on the current policy with a certain degree of randomness which deviates from such a policy.

1.2 Rewarding behaviors

In some domains, it could be of interest the study of rewards not depending on a single decision (like in MDPs) but depending on a *sequence* of visited states and actions. For instance, we can reward an agent not only by reaching a goal state, but if the goal is reached while satisfying other properties of interest during the simulation or, in other words, if the agent satisfies some target behaviour. It is clear that the definition of MDP does not fit this problem since the optimal decision in the current state depends on the *history of states* that leads to the current state.

This idea of rewarding behaviours has been proposed in (Bacchus et al., 1996), by defining the *Non-Markovian Reward Decision Process* (NMRDP), a variant of an MDP where the reward does not depend only from one transition of the environment but from a sequence of transitions. In order to specify the desired behaviours that the agent should learn, they defined a temporal logic formalisms called PLTL (Past LTL), which is able to speak about a sequence of property configurations over time, that we call *traces*. The classic reinforcement learning algorithms does not work on an NMRDP; however, they propose a transformation from NMRDP into a *expanded* MDP such that the solution of the MDP is also a solution of the original problem. Hence, in order to solve the NMRDP, we can run off-the-shelf RL algorithms over the transformed MDP; the learnt Markovian policy can be easily converted into an optimal policy for the NMRDP.

The trick here is that the transformed MDP is defined over a *expanded state space*, which still contains the original state space but enriches it by labelling every state. The idea is that the labels should keep track in some way the (partial) satisfaction of the temporal formulas. As a result, every state in the transformed state space is replicated multiple times, marking the difference between different (relevant) histories terminating in that state.

A similar transformation has been done in the following works: (Thiébaux et al., 2006) and (Gretton, 2014), where the temporal logic formalism was respectively \$FLTL (a finite LTL with future formulas) and \$*FLTL (a variant of \$FLTL); in (Icarte et al., 2018), where they used *Co-Safe* LTL formulas (Kupferman and Y. Vardi, 2001; Lacerda et al., 2015); in (Camacho et al., 2017a,b), where they used LTL$_f$ (Linear Temporal Logic over finite traces) (De Giacomo and Vardi, 2013). In (Brafman et al., 2018) the specification of temporal goals is done by LTL$_f$ or LDL$_f$ (Linear Dynamic Logic over finite traces) formulas.

The construction shown in (Brafman et al., 2018) is the foundation for this work.

1.3 Goals

The idea to define non-Markovian rewards is interesting and, as we've seen in the previous section, has become popular. However, the cited works are mainly focused on *planning over NMRDP*, where the model is known (i.e. the transition function and the reward function).

Instead, in this work we focused on reinforcement learning over NRMDP, specified by LTL$_f$/LDL$_f$ formulas; that is, we assume nothing about the transition function and the reward function, but we aim to let the agent learn how to accomplish the temporal goals. The agent, differently from the planning domain, during the simulations, is in charge of doing actions, observing the new states and collecting rewards, without prior knowledge. It has to come up with a policy learnt from experience.

Our first goal is to give theoretical foundations for reinforcement learning for non-Markovian rewards expressed by LTL$_f$/LDL$_f$ formulas. LDL$_f$ formalism is expressive as Monadic Second-Order logic (MSO), allowing to express any kind of complex temporally extended goal.

Then, we explore the possibility to have two different separated representations of the world: one used by the agent where it learns; the other to talk about temporally extended goals in LTL$_f$/LDL$_f$. This construction has several advantages, such as modularity of the feature design and a reduced state space used by the learning agent.

Due to the *sparsity of rewards* (i.e. the reward is given to the agent only when the goal is reached), we should devise a way to help exploration of the state space, by ignoring the non-relevant portion of the state space. This is a crucial step for speed-up the convergence rate of the learning process.

An important goal of this work is to look for an implementation of the theoretical settings explored, as well as an experimental evidence, to prove that our approach actually makes sense. In particular, we need a way to manage LTL$_f$/LDL$_f$ formalisms and the reinforcement learning for LTL$_f$/LDL$_f$ goals to let the agent learn temporal goals.

1.4 Achievements

The first contribution of this thesis is to devise a new algorithm for the translation from LTL$_f$/LDL$_f$ formulas to DFA, without the need to first translate into a NFA and then determinize it. We call it LDL$_f$2DFA. Actually, it is a variant of the LDL$_f$2NFA algorithm described in (Brafman et al., 2018), but with the advantage to generate only the reachable states of the DFA, which in practice yields better performances.

Then, we created the Python package FLLOAT (From LTL$_f$ and LDL$_f$ tO auTomata) that deals with the transformation from LTL$_f$/LDL$_f$ formulas to equivalent automata. This is a crucial step for the computation of the extended MDP from an NMRDP with LTL$_f$/LDL$_f$ rewards (Brafman et al., 2018).

We formalized the problem of *RL over NMRDP with* LTL$_f$/LDL$_f$, by leveraging the result in (Brafman et al., 2018). We reduce the problem to classical RL over an equivalent

MDP, hence enabling the use of off-the-shelf RL algorithms to find a solution to the original problem.

Then, we focused on a two-fold representation of the world. As stated before, one is the agent's representation that the agent uses to learn. The other one is used for specifying temporally extended goals expressed in LTL$_f$/LDL$_f$. We give motivations about this approach and analyzed the implications and the main advantages. Hence, we defined a new problem, *RL for* LTL$_f$/LDL$_f$ *goals*, and we provided a solution that reduces the problem to reinforcement learning over an MDP (analogously to what we did for *RL over NMRDP*).

We devised a way to apply *reward shaping* to this setting, by leveraging the particular structure of our solution. Reward shaping is a well-known technique in reinforcement learning to speed up the convergence rate and to help the agent to explore the state space more efficiently. In our setting, we shaping-reward the transitions that make a step toward the satisfaction of the LTL$_f$/LDL$_f$ goals φ; this reduces to reward transitions where the progression of the associated automata $\mathcal{A}_\varphi$ become nearer to an accepting state. We design two techniques. One that requires the $\mathcal{A}_\varphi$ to be built, i.e. is known before the learning process, that we call *off-line* reward shaping. The other, instead, builds the automaton from scratch, by observing the transitions made during the learning process, that we call *on-the-fly* reward shaping.

As a practical contribution, we implemented RLTG (Reinforcement Learning for Temporal Goal), a Python framework for easy set up a reinforcement learning experiment with LTL$_f$/LDL$_f$ goals, by leveraging the above-mentioned FLLOAT for the construction of the automata. It works both as a classic reinforcement learning framework and a LTL$_f$/LDL$_f$ goal-based framework.

Finally, we designed experiments with some simulated RL environments and implemented them with FLLOAT and RLTG. We introduced LTL$_f$/LDL$_f$ goals to classic RL environments (e.g. BREAKOUT), and show that the agent learns a policy that accomplishes those goals. In other words, they provided experimental evidence of the goodness and practicability of our construction.

1.5 Structure of the Thesis

The rest of the thesis is structured as follows:

- In Chapter 2 we describe the notions of temporal logic formalisms, that will be used for temporal goal specifications. We start from LTL, RE and then we move towards LTL$_f$ and LDL$_f$, upon which our method is built on. Moreover, we describe a new algorithm for the conversion of LTL$_f$/LDL$_f$ formulas to equivalent automata;
- In Chapter 3 we describe FLLOAT, a software project that implements the translation from LTL$_f$/LDL$_f$ formulas to equivalent automata. Such translation is an important piece of our approach and for the implementations of RL experiments;
- In Chapter 4 we introduce foundational concepts in reinforcement learning, MDPs and algorithm to find a solution. Then we move to NMRDPs and we explain describe how to do RL for NMRDP with LTL$_f$/LDL$_f$ goals;
- In Chapter 5 the two-fold representation of the world, the agent's low-level one where the agent learns and the high-level one used for express LTL$_f$/LDL$_f$ formulas,

is introduced. We discuss implications and the advantages of our approach, and formally define a new problem. We also describe a solution to the problem.

- In Chapter 6 we apply reward shaping techniques to the setting explained in Chapter 4. We propose an automata-based reward shaping, in the sense that we based the positive/negative extra rewards by the transitions over the automata, which track the satisfaction of the $\text{LTL}_f/\text{LDL}_f$ goals;
- In Chapter 7 we present a reinforcement learning framework allowing easy implementation of the construction described in Chapter 4 and 6.
- Chapter 8 describes the conducted experiments,5 giving evidence that our approach actually works.
- The thesis is concluded in Chapter 9. This chapter summarizes also the achievements of the thesis and discusses future works.

Part of this work has been submitted to a top AI conference and it is under review. You can find it at https://arxiv.org/abs/1807.06333.

Chapter 2

LTL_f and LDL_f

In this chapter, we introduce the reader to the main important framework to talk about behaviours over time, which gives the foundations for our approach. First, we talk about the well-known Linear-Time Temporal Logic (LTL), Propositional Dynamic Logic (PDL) and their main applications; then we go more in deep by presenting a specific formalism, namely *Linear Temporal Logic over Finite Traces* LTL_f and *Linear Dynamic Logic over Finite Traces* LDL_f. We describe an algorithm to transform an $\text{LTL}_f/\text{LDL}_f$ formula to a NFA automaton that allow to reasoning about the formula. Finally, we study the translation from $\text{LTL}_f/\text{LDL}_f$ formulas to Deterministic Finite Automata (DFA), that is a variant of the original algorithm. In this chapter have been provided many examples in order to better understand the theoretical topics. We require the reader to be acquainted with classical logic (Shapiro and Kouri Kissel, 2018) and automata theory (Hopcroft et al., 2000).

2.1 Linear-time Temporal Logic (LTL)

Temporal Logic (Goranko and Galton, 2015) is a category of formal languages aimed to talk about properties of a system whose truth value might change over time. This is in contrast with atemporal logics, which can only discuss statements whose truth value is constant.

Linear time Temporal Logic (Pnueli, 1977), or *Linear Temporal Logic* (LTL) is such a logic. It is the most popular and widely used temporal logic in computer science, especially in formal verification of software/hardware systems, in AI to reasoning about actions and planning, and in the area of Business Process Specification and Verification to specify processes declaratively.

It allows to express temporal patterns about some property p, like *liveness* (*p will eventually happen*), *safety* (*p will never happen*) and *fairness*, combinations of the previous patterns (*infinitely often p holds*, *eventually always p holds*).

2.1.1 Syntax

A LTL formula φ is defined over a set of propositional symbols $\mathcal{P}$ and are closed under the boolean connectives, the unary temporal operator $\circ$(*next-time*) and the binary operator $\mathcal{U}$ (*until*):

$$\varphi \ ::= \ A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \circ\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

With $A \in \mathcal{P}$.

Additional operators can be defined in terms of the ones above: as usual logical operators such as $\vee, \Rightarrow, \Leftrightarrow$, *true*, *false* and temporal formulas like *eventually* as $\Diamond\varphi \doteq true\,\mathcal{U}\,\varphi$, *always* as $\Box\varphi \doteq \neg\Diamond\neg\varphi$ and *release* as $\varphi_1 \mathcal{R} \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$.

Example 2.1. Several interesting temporal properties can be defined in LTL:

- *Liveness*: $\Diamond\varphi$, which means "condition expressed by φ *at some time* in the future will be satisfied", "sooner or later φ will hold" or "eventually φ will hold". E.g., $\Diamond rich$ (eventually I will become rich), $Request \implies \Diamond Response$ (if someone requested the service, sooner or later he will receive a response).
- *Safety*: $\Box\varphi$, which means "condition expressed by φ, *every time* in the future will be satisfied", "always φ will hold". E.g., $\Box happy$ (I'm always happy), $\Box\neg(temperature > 30)$ (the temperature of the room must never be over 30).
- *Response*: $\Box\Diamond\varphi$ which means "at any instant of time there exists a moment later where φ holds". This temporal pattern is known in computer science as *fairness*.
- *Persistence*: $\Diamond\Box\varphi$, which stand for "There exists a moment in the future such that from then on φ always holds". E.g. $\Diamond\Box dead$ (at a certain point you will die, and you will be dead forever)
- *Strong fairness*: $\Box\Diamond\varphi_1 \implies \Box\Diamond\varphi_2$, "if something is attempted/requested infinitely often, then it will be successful/allocated infinitely often". E.g., $\Box\Diamond ready \implies \Box\Diamond run$ (if a process is in ready state infinitely often, then infinitely often it will be selected by the scheduler).

2.1.2 Semantics

The semantics of LTL is provided by (infinite) *traces*, i.e. ω-word over the alphabet $2^{\mathcal{P}}$.

Definition 2.1. Given a infinite trace π, we define that a LTL formula φ is *true* at time i, in symbols $\pi, i \models \varphi$ inductively as follows:

$$\pi, i \models A, \text{ for } A \in \mathcal{P} \text{ iff } A \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$

$$\pi, i \models \circ\varphi \text{ iff } \pi, i+1 \models \varphi$$

$$\pi, i \models \varphi_1 \mathcal{U} \varphi_2 \text{ iff } \exists j.(j \geq i) \wedge \pi, j \models \varphi \wedge \forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1$$

Similarly as in classical logic we give the following definitions:

Definition 2.2. A LTL formula is *true* in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is *satisfiable* if it is true in some π and is *valid* if it is true in every π. φ_1 *entails* φ_2, in symbols $\varphi_1 \models \varphi_2$ iff $\forall \pi, \forall i. \pi, i \models \varphi_1 \implies \pi, i \models \varphi_2$.

Now we state an important result:

Theorem 2.1 (Sistla and Clarke (1985))**.** *Satisfiability, validity, and entailment for* LTL *formulas are* PSPACE*-complete.*

Indeed, Linear Temporal Logic can be thought of as a specific decidable (PSPACE-complete) fragment of classical first-order logic (FOL).
Notice that, a *trace* π can be seen as a *word* on a *path* of a *Kripke structure*.

Definition 2.3 (Clarke et al. (1999))**.** a Kripke structure $\mathcal{K}$ over a set of propositional symbols $\mathcal{P}$ is a 4-tuple $\langle S, I, R, L \rangle$ where S is a finite set of *states*, $I \subseteq S$ is the set of *initial states*, $R \subseteq S \times S$ is the *transition relation* such that R is left-total and $L : S \to 2^{\mathcal{P}}$ is a *labeling function.*

A *path* ρ over $\mathcal{K}$ is a sequence of states $\langle s_1, s_2, \ldots \rangle$ such that $\forall i. R(s_i, s_{i+1})$. From a path we can build a *word* w on the path ρ by mapping each state of the sequence with L, namely:

$$w = \langle L(s_1), L(s_2), \ldots \rangle$$

In simpler words, a trace of propositional symbols $\mathcal{P}$ is a infinite sequence of combinations of propositional symbols in $\mathcal{P}$. Moreover, we denote by $\pi(i)$ with $i \in \mathbb{N}$ the labels associated to s_i, i.e. $L(s_i)$.

Example 2.2. In figure 2.1 is depicted an example of Kripke structure $\mathcal{K}$ over $\mathcal{P} = \{p, q\}$ where:

$$\begin{aligned} S &= \{s_1, s_2, s_3\} \\ I &= \{s_1\} \\ R &= \{(s_1, s_2), (s_2, s_1), (s_2, s_3), (s_3, s_3)\} \\ L &= \{(s_1, \{p, q\}), (s_2, \{q\}), (s_3, \{p\})\} \end{aligned}$$

The path $\langle s_1, s_2, s_3, s_3, s_3 \ldots \rangle$ yields the following trace π:

$$\begin{aligned} \pi &= \langle L(s_1), L(s_2), L(s_3), L(s_3), L(s_3), \ldots \rangle \\ &= \langle \{p, q\}\{q\}, \{p\}, \{p\}, \{p\}, \ldots \rangle \end{aligned}$$

2.2 Linear Temporal Logic on Finite Traces: LTL$_f$

Linear-time Temporal Logic over finite traces, LTL$_f$, is essentially standard LTL (Pnueli, 1977) interpreted over finite, instead of over infinite, traces (De Giacomo and Vardi, 2013). This apparently trivial difference has a big impact: as we will see, some LTL formula has a different meaning if interpreted over infinite traces or finite ones.

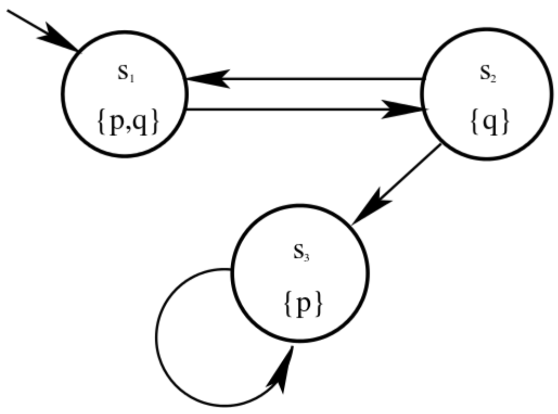


Figure 2.1. An example of Kripke structure.

2.2.1 Syntax

In fact, the syntax of LTL$_f$ is the same of the one showed in Section 2.1.1, i.e. *formulas* of LTL$_f$ are built from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\circ$(*next-time*) and the binary operator $\mathcal{U}$ (*until*):

$$\varphi \quad ::= \quad A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \circ\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

With $A \in \mathcal{P}$.

We use the standard abbreviations for classical logic formulas:

$$\begin{aligned}
\varphi_1 \vee \varphi_2 &\doteq \neg(\neg\varphi_1 \wedge \neg\varphi_2)\\
\varphi_1 \Rightarrow \varphi_2 &\doteq \neg\varphi_1 \vee \varphi_2\\
\varphi_1 \Leftrightarrow \varphi_2 &\doteq \varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1\\
\mathit{true} &\doteq \neg\varphi \vee \varphi\\
\mathit{false} &\doteq \neg\varphi \wedge \varphi
\end{aligned}$$

And for temporal formulas:

$$\varphi_1 \mathcal{R} \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2) \tag{2.1}$$

$$\Diamond\varphi \doteq \mathit{true}\,\mathcal{U}\,\varphi \tag{2.2}$$

$$\Box\varphi \doteq \neg\Diamond\neg\varphi \tag{2.3}$$

$$\bullet\varphi \doteq \neg\circ\neg\varphi \tag{2.4}$$

$$\mathit{Last} \doteq \bullet\mathit{false} \tag{2.5}$$

$$\mathit{End} \doteq \Box\mathit{false} \tag{2.6}$$

As the reader might already noticed, 2.2 and 2.3 are defined as in Section 2.1.1; Equation 2.1 is called *release*; Equation 2.4 is called *weak next* (notice that on finite traces $\neg\circ\varphi \not\equiv \circ\neg\varphi$); 2.5 denotes the end of the trace, while 2.6 denotes that the trace is ended.

Example 2.3. Here we recall Example 2.1 and we see the impact on *Always*, *Eventually Response* and *Persistence* LTL formulas if interpreted on finite traces (i.e. formulas in LTL$_f$):

- *Safety*: $\Box A$ means that always *till the end of the trace* φ holds;
- *Liveness*: $\Diamond A$ means that eventually *before the end of the trace* φ holds;
- *Response*: $\Box\Diamond\varphi$ on finite traces becomes equivalent to *last point in the trace satisfies* φ, i.e. $\Diamond(Last \wedge \varphi)$. Intuitively, this is true because $\Box\Diamond\varphi$ implies that at the last point in the trace φ holds (because there are no successive instants of time that make φ true); but if this is the case, then what happens at previous points in the trace does not matter because the formula evaluates always to true, since as we just said φ must hold at the last point in the trace, hence the equivalence with $\Diamond(Last \wedge \varphi)$.
- *Persistence*: $\Diamond\Box\varphi$ on finite traces becomes equivalent to *last point in the trace satisfies* φ, i.e. $\Diamond(Last \wedge \varphi)$. Analogously to the previous case, the equivalence holds because $\Diamond\Box\varphi$ implies that at the last point in the trace $\Box\varphi$ holds (and so φ) since we have no further successive instants of time that make $\Box\varphi$ true. But if this is the case, then what happens at previous points in the trace does not matter because the formula evaluates always to true, since as we just said $\Box\varphi$ (and so φ) must hold at the last point in the trace, hence the equivalence with $\Diamond(Last \wedge \varphi)$.

In other words, no direct nesting of eventually and always connectives is meaningful in LTL$_f$, and this contrast what happens in LTL of infinite traces.

Example 2.4. Another remarkable evidence about the relevance of the assumption about the finiteness of traces is provided by the DECLARE approach (Pesic and van der Aalst, 2006).

DECLARE is a declarative approach to business process modeling based on LTL interpreted over finite traces. The intuition is to map finite traces describing a domain of interest (e.g. processes) into infinite traces under the assumption that

$$\Diamond end \wedge \Box(end \Rightarrow \circ end) \wedge \Box(end \Rightarrow \bigwedge_{p\in\mathcal{P}} \neg p) \tag{2.7}$$

which means that the following english statements hold:

- *end* eventually holds ($end \notin \mathcal{P}$);
- once *end* is true, it is true forever;
- when *end* is true all other propositions must be false

In other words, every finite trace π_f is extended with an infinite sequence of *end*, or in symbols $\pi_{inf} = \pi_f\{end\}^\omega$. By construction we have that

$$\pi_{inf} \models \Diamond end \wedge \Box(end \Rightarrow \circ end) \wedge \Box(end \Rightarrow \bigwedge_{p\in\mathcal{P}} \neg p)$$

Despite it seems a nice construction to adapt LTL on finite traces, in fact it is wrong due to the *next* operator: in an infinite trace a successor state always exists, whereas in a finite one this does not hold. There exists a counterexample showing that the interpretation of LTL formulas on finite traces with the construction just explained is **not** equivalent with proper interpretation over finite traces offered by LTL$_f$, i.e. in general:

$$\pi_f\{end\}^\omega \models \varphi \not\Leftrightarrow \pi_f \models_f \varphi \tag{2.8}$$

To see why this is the case, consider the DECLARE "negation chain succession" $\Box(a \Rightarrow \circ\neg b)$ which requires that at any point in the trace, the state after we see a, b is false. Consider also the finite trace $\pi_f = \{a\}$ and the associated infinite trace $\pi_{inf} = \{a\}\{end\}^\omega$ built as explained before. We have that

$$\pi_{inf} \models \Box(a \Rightarrow \circ\neg b)$$

where $\models$ has been defined in 2.1. This is true because there is only one occurrence of a and then *end* holds forever (and so b does not).

But if the same formula is interpreted on finite traces (namely $\models_f$):

$$\pi_f \not\models_f \Box(a \Rightarrow \circ\neg b)$$

because the finite trace a is true at the last instant, but then there is no next instance where b is false, so $\circ\neg b$ is evaluated to *false* and the formula does not hold. The correct way to express "negation chain succession" on finite traces would be $\Box(a \Rightarrow \bullet\neg b)$.

The LTL formulas φ that are insensitive to the problem just shown, i.e. such that

$$\pi_f\{end\}^\omega \models \varphi \text{ iff } \pi_f \models_f \varphi \tag{2.9}$$

holds are defined *insensitive to infiniteness* (De Giacomo et al., 2014). This is another important evidence about the the relevance of the finiteness trace assumption.

2.2.2 Semantics

Formally, a *finite trace* π is a finite word over the alphabet $2^{\mathcal{P}}$, i.e. as alphabet we have all the possible interpretations of the propositional symbols in $\mathcal{P}$. We can see π as a *finite* word on a path of a Kripke structure, similarly as we discussed in Section 2.1.2 (but in that case the traces were *infinite*). Given a finite path $\rho = \langle s_1, s_2, \ldots, s_n\rangle$ on a Kripke structure $\mathcal{K}$, a finite trace π associated to the path ρ is defined as $\langle L(s_1), L(s_2), \ldots, L(s_n)\rangle$.

We use the following notation. We denote the *length* of a trace π as $length(\pi)$. We denote the i_{th} *position* on the trace as $\pi(i) = L(s_i)$, i.e. the propositions that hold in the i_{th} state of the path, with $0 \leq i \leq last$ where $last = length(\pi) - 1$ is the last element of the trace. We denote by $\pi(i, j)$, the *segment* of π, the trace $\pi' = \langle \pi(i), \pi(i+1), \ldots, \pi(j)\rangle$, with $0 \leq i \leq j \leq last$

Definition 2.4. Given a finite trace π, we define that a LTL$_f$ formula φ is *true* at time i $(0 \leq i \leq \mathit{last})$, in symbols $\pi, i \models \varphi$ inductively as follows:

$$\pi, i \models A, \text{ for } A \in \mathcal{P} \text{ iff } A \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$

$$\pi, i \models \circ\varphi \text{ iff } i < \mathit{last} \wedge \pi, i+1 \models \varphi \tag{2.10}$$

$$\pi, i \models \varphi_1 \mathcal{U} \varphi_2 \text{ iff } \exists j.(i \leq j \leq \mathit{last}) \wedge \pi, j \models \varphi \wedge$$

$$\forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1 \tag{2.11}$$

Notice that Definition 2.4 is pretty similar to Definition 2.1, except the bounding of indexes in Equation 2.10 and Equation 2.11, to recognize that the trace is ended.

Analogously to Definition 2.2 we give the following definitions:

Definition 2.5. A LTL$_f$ formula is *true* in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is *satisfiable* if it is true in some π and is *valid* if it is true in every π. φ_1 *entails* φ_2, in symbols $\varphi_1 \models \varphi_2$ iff $\forall \pi, \forall i. \pi, i \models \varphi_1 \implies \pi, i \models \varphi_2$.

2.2.3 Complexity and Expressiveness

Thanks to reduction of LTL$_f$ satisfiability (Definition 2.5) into LTL satisfiability for PSPACE membership and reduction of STRIPS planning into LTL$_f$ satisfiability for PSPACE-hardness, as proposed in (De Giacomo and Vardi, 2013), we have this result:

Theorem 2.2 (De Giacomo and Vardi (2013))**.** *Satisfiability, validity and entailment for* LTL$_f$ *formulas are* PSPACE*-complete.*

About expressiveness of LTL$_f$, we have that:

Theorem 2.3 (De Giacomo and Vardi (2013); Gabbay et al. (1997))**.** LTL$_f$ *has exactly the same expressive power of* FOL *over finite ordered sequences.*

2.3 Regular Temporal Specifications (RE$_f$)

In this section, we talk about regular languages as a form of temporal specification over finite traces. In particular we focus on regular expressions (Hopcroft et al., 2000).

A regular expression ϱ is defined inductively as follows, considering as alphabet the set of propositional interpretations $2^{\mathcal{P}}$, from a set of propositional symbols $\mathcal{P}$:

$$\varrho \quad ::= \quad \phi \mid \varrho_1 + \varrho_2 \mid \varrho_1 ; \varrho_2 \mid \varrho^*$$

where ϕ is a propositional formula that is an abbreviation for the union of all the propositional interpretations that satisfy ϕ, i.e. $\phi = \sum_{\Pi \models \phi} \Pi$ and $\Pi \in 2^{\mathcal{P}}$.

We denote by $\mathcal{L}(\varrho)$ the language recognized by a RE$_f$ expression. We interpret these expressions over finite traces, introduced in Section 2.2.2.

Definition 2.6. We say that a regular expression ϱ *is true* in the finite trace π ifs $\pi \in \mathcal{L}(\varrho)$. We say that ϱ *is true at instant* i if $\pi(i, last) \in \mathcal{L}(\varrho)$. We say that ϱ *is true between instants* i, j if $\pi(i, j) \in \mathcal{L}(\varrho)$.

Example 2.5. We recall Example 2.2. The trace resulting from path $\langle s_1, s_2, s_3, s_3, \ldots\rangle$, i.e.:

$$\pi = \langle \{p, q\}\{q\}, \{p\}, \{p\}, \{p\}, \ldots\rangle$$

belongs to the language generated by the following regular expression:

$$\varrho_1 = p \wedge q; q; p^*$$

But also by this one:

$$\varrho_2 = true; q + p; true^*$$

Example 2.6. We can express some of the formulas shown in Example 2.3, and many others, in RE$_f$:

- *Safety*: φ^*, equivalent to $\Box\varphi$
- *Liveness*: $true^*; \varphi; true^*$, equivalent to $\Diamond\varphi$;
- *Response* and *Persistence*: as said before, when interpreted on finite traces, they are equivalent to $\Diamond(Last \wedge \varphi)$; hence, they can be rewritten in RE$_f$ as $true^*; \varphi$
- *Ordered occurrence*: $true^*; \varphi_1; true^*; \varphi_2; true^*$, equivalent to $\Diamond(\varphi_1 \wedge \circ\Diamond\varphi_2)$ means that φ_1 and φ_2 happen in order;
- *Alternating sequence*: $(\psi, \varphi)^*$ means that ψ and φ alternate from the beginning of the trace, starting with ψ and ending with φ.

The *Alternating sequence* is an example of formula that has not a counterpart in LTL$_f$. More generally, LTL$_f$ (and LTL) are not able to capture regular structural properties on path (Wolper, 1981).

This observation about expressiveness of RE$_f$ is confirmed by Theorem 6 of (De Giacomo and Vardi, 2013), which is a consequence of several classical results (Büchi, 1960; Elgot, 1961; Trakhtenbrot, 1961; Thomas, 1979):

Theorem 2.4 (De Giacomo and Vardi (2013))**.** *RE$_f$ is strictly more expressive than LTL$_f$*

More precisely, RE$_f$ is expressive as *monadic second-order logic* MSO over bounded ordered sequences (Khoussainov and Nerode, 2001).

2.4 Linear Dynamic Logic on Finite Traces: LDL$_f$

The problem with RE$_f$ is that, although is strictly more expressive than LTL$_f$, is considered a low-level formalism for temporal specifications. For instance, RE$_f$ misses a direct construct for negation and for conjunction. Moreover, negation requires an exponential blow-up, hence adding complementation and intersection constructs are not advisable.

Linear Dynamic Logic of Finite Traces LDL$_f$ (De Giacomo and Vardi, 2013) merges LTL$_f$ with RE$_f$ in a very natural way, borrowing the syntax of PDL (Fischer and Ladner, 1979), a well-known (propositional) logic of programs in computer science. It keeps the declarativeness and convenience of LTL$_f$ while having the same expressive power of RE$_f$.

2.4.1 Syntax

Formally, LDL$_f$ formulas φ are built over a set of propositional symbols $\mathcal{P}$ as follows (Brafman et al., 2017):

$$\begin{array}{lcl}\varphi & ::= & tt \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\varrho\rangle\varphi \\ \varrho & ::= & \phi \mid \varphi? \mid \varrho_1 + \varrho_2 \mid \varrho_1;\varrho_2 \mid \varrho^*\end{array}$$

where tt stands for logical true; ϕ is a propositional formula over $\mathcal{P}$; ϱ denotes path expressions, which are RE over propositional formulas ϕ with the addition of the test construct $\varphi?$ typical of PDL. Moreover, we use the following abbreviations for classical logic operators:

$$\begin{aligned}\varphi_1 \vee \varphi_2 &\doteq \neg(\neg\varphi_1 \wedge \neg\varphi_2)\\ \varphi_1 \Rightarrow \varphi_2 &\doteq \neg\varphi_1 \vee \varphi_2\\ \varphi_1 \Leftrightarrow \varphi_2 &\doteq \varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1\\ ff &\doteq \neg tt\end{aligned}$$

And for temporal formulas:

$$[\varrho]\varphi \doteq \neg\langle\varrho\rangle\neg\varphi \tag{2.12}$$

$$End \doteq [true]ff \tag{2.13}$$

$$Last \doteq \langle true\rangle End \tag{2.14}$$

$[\varrho]\varphi$ and $\langle\varrho\rangle\varphi$ are analogous to box and diamond operators in PDL; Formula 2.14 denotes the last element of the trace, whereas Formula 2.13 denotes that the trace is ended. Intuitively, $\langle\varrho\rangle\varphi$ states that, from the current step in the trace, there exists an execution satisfying the RE ϱ such that its last step satisfies φ, while $[\varrho]\varphi$ states that, from the current step, all executions satisfying the RE ϱ are such that their last step satisfies φ.

2.4.2 Semantics

As we did in the previous sections, we formally give a semantics to LDL$_f$ (interpreted over finite traces, like LTL$_f$ and RE).

Definition 2.7. Given a finite trace π, we define that a LDL$_f$ formula φ is *true* at time i ($0 \leq i \leq last$), in symbols $\pi, i \models \varphi$ inductively as follows:

$\pi, i \models tt$
$\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$
$\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$
$\pi, i \models \langle\phi\rangle\varphi$ iff $i < last \wedge \pi(i) \models \phi \wedge \pi, i+1 \models \varphi$
$\pi, i \models \langle\psi?\rangle\varphi$ iff $\pi, i \models \psi \wedge \pi, i \models \varphi$
$\pi, i \models \langle\varrho_1 + \varrho_2\rangle\varphi$ iff $\pi, i \models \langle\varrho_1\rangle\varphi \vee \langle\varrho_2\rangle\varphi$
$\pi, i \models \langle\varrho_1;\varrho_2\rangle\varphi$ iff $\pi, i \models \langle\varrho_1\rangle\langle\varrho_2\rangle\varphi$
$\pi, i \models \langle\varrho^*\rangle\varphi$ iff $\pi, i \models \varphi \vee i < last \wedge \pi, i \models \langle\varrho\rangle\langle\varrho^*\rangle\varphi$ and ϱ is not *test-only*

We say that ϱ is *test-only* if it is a RE$_f$ expression whose atoms are only tests, i.e. $\psi?$.

Notice that LDL$_f$ fully captures LTL$_f$. For every formula in LTL$_f$ there exists a LDL$_f$ formula with the same meaning, namely:

LTL$_f$	LDL$_f$
A	$\langle A\rangle tt$
$\neg\varphi$	$\neg\varphi$
$\varphi_1 \wedge \varphi_2$	$\varphi_1 \wedge \varphi_2$
$\circ\varphi$	$\langle true\rangle(\varphi \wedge \neg End)$
$\varphi_1 \mathcal{U}\, \varphi$	$\langle(\varphi_1?; true)^*\rangle(\varphi_2 \wedge \neg End)$

Notice also that every RE$_f$ expression ϱ is captured in LDL$_f$ by $\langle\varrho\rangle End$. Moreover, since also the converse holds, i.e. every LDL$_f$ formula can be expressed in RE (by Theorem 11 in (De Giacomo and Vardi, 2013)), the following theorem holds:

Theorem 2.5 (De Giacomo and Vardi (2013))**.** LDL$_f$ *has exactly the same expressive power of* MSO

Now we show several LDL$_f$ examples.

Example 2.7. Formulas described in Examples 2.3 and 2.6 can be rewritten in LDL$_f$ as:

- *Safety*: $[true^*]\varphi$, equivalent to LTL$_f$ formula $\Box\varphi$
- *Liveness*: $\langle true^*\rangle\varphi$, equivalent to LTL$_f$ formula $\Diamond\varphi$
- *Conditional Response*: $[true^*](\varphi_1 \Rightarrow \langle true^*\rangle\varphi_2)$, equivalent to LTL$_f$ formula $\Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$
- *Ordered occurrence*: $\langle true^*; \varphi_1; true^*; \varphi_2; true^*\rangle End$ equivalent to the RE$_f$ expression $true^*; \varphi_1; true^*; \varphi_2; true^*$
- *Alternating occurrence*: $\langle(\psi; \varphi)^*\rangle End$ equivalent to the RE$_f$ expression $(\psi; \varphi)^*$

Example 2.8. Consider the Example 2.2 and 2.5. ϱ_1 and ϱ_2 are translated into LDL$_f$ as $\langle\varrho_1\rangle End$ and $\langle\varrho_2\rangle End$ respectively.
Other LDL$_f$ formulas satisfiable in the Kripke structure $\mathcal{K}$ depicted in Figure 2.1 are:

- $\langle p\rangle tt$ by every (non-empty) path, since s_1 is the initial state and we have that $\{p, q\} \models p$
- $\langle q\rangle tt$ as the previous case
- $\langle(p; q); (p; q)^*; p; p^*\rangle tt$ by paths of the form $\rho = s_1, s_2, (s_1, s_2)^\omega, s_3, (s_3)^\omega$
- $[true^*]\langle p \vee q\rangle tt$ is satisfied for every path, since for every reachable state either p or q are true;

2.5 LTL$_f$ and LDL$_f$ translation to automata

Given an LTL$_f$/LDL$_f$ formula φ, we can construct a deterministic finite state automaton (DFA) (Rabin and Scott, 1959) $\mathcal{A}_\varphi$ that accept the same finite traces that makes φ true. In order to do this, we proceed in two steps: First we translate LTL$_f$ and LDL$_f$ formulas into (NFA) (De Giacomo and Vardi, 2015). Then the NFA obtained can be transformed into a DFA following the standard procedure of *determinization.*

Now we recall definitions of NFA and DFA:

Definition 2.8. An NFA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where:

- Σ is the input alphabet;
- Q is the finite set of states;
- $q_0 \in Q$ is the initial state;
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation;
- $F \subseteq Q$ is the set of final states;

Definition 2.9. A DFA is a NFA where δ is a function $\delta : Q \times \Sigma \to Q$

By $\mathcal{L}(A)$ we mean the set of all traces over Σ accepted by $\mathcal{A}$.

In the next two subsections we give some definition that will be used in the algorithm; then we describe the algorithm for the translation and give some example.

2.5.1 ∂ function for LTL$_f$

We give the following definition:

Definition 2.10. The *delta function ∂ for LTL$_f$ formulas* is a function that takes as input an (implicitly quoted) LTL$_f$ formula φ in NNF and a propositional interpretation Π for $\mathcal{P}$, and returns a positive boolean formula whose atoms are (implicitly quoted) φ

subformulas. It is defined as follows:

$$\begin{aligned}
\partial(A,\Pi) &= \begin{cases} \mathit{true} & \text{if } A \in \Pi \\ \mathit{false} & \text{if } A \notin \Pi \end{cases} \\
\partial(\neg A,\Pi) &= \begin{cases} \mathit{false} & \text{if } A \in \Pi \\ \mathit{true} & \text{if } A \notin \Pi \end{cases} \\
\partial(\varphi_1 \wedge \varphi_2,\Pi) &= \partial(\varphi_1,\Pi) \wedge \partial(\varphi_2,\Pi) \\
\partial(\varphi_1 \vee \varphi_2,\Pi) &= \partial(\varphi_1,\Pi) \vee \partial(\varphi_2,\Pi) \\
\partial(\circ\varphi,\Pi) &= \varphi \wedge \neg \mathit{End} \equiv \varphi \wedge \Diamond \mathit{true} \\
\partial(\varphi_1 \mathcal{U} \varphi_2,\Pi) &= \partial(\varphi_2,\Pi) \vee (\partial(\varphi_1,\Pi) \wedge \partial(\circ(\varphi_1 \mathcal{U} \varphi_2),\Pi)) \\
\partial(\bullet\varphi,\Pi) &= \varphi \vee \mathit{End} \equiv \varphi \vee \Box \mathit{false} \\
\partial(\varphi_1 \mathcal{R} \varphi_2,\Pi) &= \partial(\varphi_2,\Pi) \wedge (\partial(\varphi_1,\Pi) \vee \partial(\bullet(\varphi_1 \mathcal{R} \varphi_2),\Pi))
\end{aligned} \tag{2.15}$$

where *End* is defined as Equation 2.6. As a consequence of Definition 2.10 and from Equation 2.2 and 2.3, we can deduce that

$$\begin{aligned}
\partial(\Diamond\varphi,\Pi) &= \partial(\varphi,\Pi) \vee \partial(\circ\Diamond\varphi,\Pi) \\
\partial(\Box\varphi,\Pi) &= \partial(\varphi,\Pi) \wedge \partial(\bullet\Box\varphi,\Pi)
\end{aligned}$$

Moreover, we define $\partial(\varphi,\epsilon)$ which is inductively defined as Equation 2.15, except for the following cases:

$$\begin{aligned}
\partial(A,\epsilon) &= \mathit{false} \\
\partial(\neg A,\epsilon) &= \mathit{false} \\
\partial(\circ\varphi,\epsilon) &= \mathit{false} \\
\partial(\bullet\varphi,\epsilon) &= \mathit{true} \\
\partial(\varphi_1 \mathcal{U} \varphi_2,\epsilon) &= \mathit{false} \\
\partial(\varphi_1 \mathcal{R} \varphi_2,\epsilon) &= \mathit{true}
\end{aligned} \tag{2.16}$$

Note that $\partial(\varphi,\epsilon)$ is always either *true* or *false*. It is worth to observe for future use that from Equation 2.16 we can say $\partial(\Diamond\varphi,\epsilon) = \mathit{false}$ and $\partial(\Box\varphi,\epsilon) = \mathit{true}$.

2.5.2 ∂ function for LDL$_f$

We give the following definition:

Definition 2.11. The *delta function ∂ for* LDL$_f$ *formulas* is a function that takes as input an (implicitly quoted) LDL$_f$ formula φ in NNF, extended with auxiliary constructs $\boldsymbol{F}_\psi$ and $\boldsymbol{T}_\psi$, and a propositional interpretation Π for $\mathcal{P}$, and returns a positive boolean formula whose atoms are (implicitly quoted) φ subformulas (not including $\boldsymbol{F}_\psi$ or $\boldsymbol{T}_\psi$). It is defined as follows:

$$\begin{array}{rcl}
\partial(tt,\Pi) & = & true\\
\partial(f\!f,\Pi) & = & false\\
\partial(\phi,\Pi) & = & \partial(\langle\phi\rangle tt,\Pi)\\
\partial(\varphi_1\wedge\varphi_2,\Pi) & = & \partial(\varphi_1,\Pi)\wedge\partial(\varphi_2,\Pi)\\
\partial(\varphi_1\vee\varphi_2,\Pi) & = & \partial(\varphi_1,\Pi)\vee\partial(\varphi_2,\Pi)\\
\partial(\langle\phi\rangle\varphi,\Pi) & = & \begin{cases}\boldsymbol{E}(\varphi) & \text{if } \Pi\models\phi\\ false & \text{if } \Pi\not\models\phi\end{cases}\\
\partial(\langle\varrho?\rangle\varphi,\Pi) & = & \partial(\varrho,\Pi)\wedge\partial(\varphi,\Pi)\\
\partial(\langle\varrho_1+\varrho_2\rangle\varphi,\Pi) & = & \partial(\langle\varrho_1\rangle\varphi,\Pi)\vee\partial(\langle\varrho_2\rangle\varphi,\Pi)\\
\partial(\langle\varrho_1;\varrho_2\rangle\varphi,\Pi) & = & \partial(\langle\varrho_1\rangle\langle\varrho_2\rangle\varphi,\Pi)\\
\partial(\langle\varrho^*\rangle\varphi,\Pi) & = & \partial(\varphi,\Pi)\vee\partial(\langle\varrho\rangle\boldsymbol{F}_{\langle\varrho^*\rangle\varphi},\Pi)\\
\partial([\phi]\varphi,\Pi) & = & \begin{cases}\boldsymbol{E}(\varphi) & \text{if } \Pi\models\phi\\ true & \text{if } \Pi\not\models\phi\end{cases}\\
\partial([\varrho?]\varphi,\Pi) & = & \partial(nnf(\neg\varrho),\Pi)\vee\partial(\varphi,\Pi)\\
\partial([\varrho_1+\varrho_2]\varphi,\Pi) & = & \partial([\varrho_1]\varphi,\Pi)\wedge\partial([\varrho_2]\varphi,\Pi)\\
\partial([\varrho_1;\varrho_2]\varphi,\Pi) & = & \partial([\varrho_1][\varrho_2]\varphi,\Pi)\\
\partial([\varrho^*]\varphi,\Pi) & = & \partial(\varphi,\Pi)\wedge\partial([\varrho]\boldsymbol{T}_{\langle\varrho^*\rangle\varphi},\Pi)\\
\partial(\boldsymbol{T}_\psi,\Pi) & = & true\\
\partial(\boldsymbol{F}_\psi,\Pi) & = & false
\end{array} \tag{2.17}$$

where $\boldsymbol{E}(\varphi)$ recursively replaces in φ all occurrences of atoms of the form $\boldsymbol{T}_\psi$ and $\boldsymbol{F}_\psi$ by $\boldsymbol{E}(\psi)$.

Moreover, we define $\partial(\varphi,\epsilon)$ which is inductively defined as Equation 2.17, except for the following cases:

$$\begin{array}{rcl}
\partial(\langle\phi\rangle\varphi,\epsilon) & = & false\\
\partial([\phi]\varphi,\epsilon) & = & true
\end{array} \tag{2.18}$$

Note that $\partial(\varphi,\epsilon)$ is always either *true* or *false*.

2.5.3 The LDL$_f$2NFA algorithm

Algorithm 2.1 (LDL$_f$2NFA) takes in input a LDL$_f$/LTL$_f$ formula φ and outputs a NFA $\mathcal{A}_\varphi=\langle 2^{\mathcal{P}},Q,q_0,\delta,F\rangle$ that accepts exactly the traces satisfying φ. It is a variant of the algorithm presented in (De Giacomo and Vardi, 2015), and its correctness relies on the fact that every LDL$_f$/LTL$_f$ formula φ can be associated a polynomial *alternating automaton on words* (AFW) accepting exactly the traces that satisfy φ and that every AFW can be transformed into an NFA (De Giacomo and Vardi, 2013). The proposed algorithm requires that φ is in *negation normal form* (NNF), i.e. with negation symbols occurring only in front of propositions.

The function ∂ used in lines 5, 12 and 15 is the one defined in sections 2.5.1 and 2.5.2; whether we are translating a LTL$_f$ or a LDL$_f$ formula, we use the function ∂ from Definition 2.10 and from Definition 2.11, respectively.

Algorithm 2.1. LDL$_f$2NFA: from LTL$_f$/LDL$_f$ formula φ to NFA $\mathcal{A}_\varphi$

```
 1: input LDL$_f$/LTL$_f$ formula $\varphi$
 2: output NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F \rangle$
 3: $q_0 \leftarrow \{\varphi\}$
 4: $F \leftarrow \{\emptyset\}$
 5: if $(\partial(\varphi, \epsilon) = true)$ then
 6:     $F \leftarrow F \cup \{q_0\}$
 7: end if
 8: $Q \leftarrow \{q_0, \emptyset\}$
 9: $\delta \leftarrow \emptyset$
10: while ($Q$ or $\delta$ change) do
11:     for $(q \in Q)$ do
12:         if $(q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi))$ then
13:             $Q \leftarrow Q \cup \{q'\}$
14:             $\delta \leftarrow \delta \cup \{(q, \Pi, q')\}$
15:             if $(\bigwedge_{(\psi \in q')} \partial(\psi, \epsilon) = true)$ then
16:                 $F \leftarrow F \cup \{q'\}$
17:             end if
18:         end if
19:     end for
20: end while
```

How LDL$_f$2NFA works

The NFA $\mathcal{A}_\varphi$ for a LDL$_f$ formula φ is built in a forward fashion. Until convergence is reached (i.e. states and transitions do not change), the algorithm visits every state q seen until now, checks for all the possible transitions from that state and collects the results, determining the next state q', the new transition (q, Π, q') and if q' is a final state. Intuitively, the delta function ∂ emulates the semantic behaviour of every LTL$_f$/LDL$_f$ subformula after seeing Π.

States of $\mathcal{A}_\varphi$ are sets of atoms (each atom is a quoted φ subformula) to be interpreted as conjunctions. The empty conjunction $\emptyset$ stands for *true*. q' is a set of quoted subformulas of φ denoting a minimal interpretation such that $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$ (notice that we trivially have $(\emptyset, p, \emptyset) \in \delta$ for every $p \in 2^{\mathcal{P}}$).

The following result holds:

Theorem 2.6 (De Giacomo and Vardi (2015))**.** *Algorithm* LDL$_f$2NFA *is correct, i.e., for every finite trace* $\pi : \pi \models \varphi$ *iff* $\pi \in \mathcal{L}(\mathcal{A}_\varphi)$*. Moreover, it terminates in at most an exponential number of steps, and generates a set of states* S *whose size is at most exponential in the size of the formula* φ*.*

In order to obtain a DFA, the NFA $\mathcal{A}_\varphi$ can be determinized in exponential time (Rabin and Scott, 1959). Thus, we can transform a LTL$_f$/LDL$_f$ formula into a DFA of double exponential size.

Example 2.9. In this example we see a run of the Algorithm 2.1 with the LTL$_f$ formula $\Box A$ (A atomic).

0. Set up:

$$\begin{aligned} q_0 &= \{\Box A\} \\ Q &= \{q_0, \emptyset\} \\ F &= \{q_0, \emptyset\} \qquad \text{(because } \partial(\Box A, \epsilon) = \partial(\mathit{false}\,\mathcal{R}\,\neg A, \epsilon) = \mathit{true}) \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset)\} \end{aligned}$$

1. Iteration: analyze $q = \{\Box A\}$

   - with $\Pi = \{A\}$ we have

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Box A, \Pi) \\ &\models \partial(A, \Pi) \wedge \partial(\bullet\Box A, \Pi) \\ &\models \mathit{true} \wedge (\text{“}\Box A\text{”} \vee \text{“}\Box \mathit{false}\text{”}) \end{aligned}$$

   Notice that $\mathit{true} \wedge (\text{“}\Box A\text{”} \vee \text{“}\Box \mathit{false}\text{”})$ is a *propositional formula* with LTL$_f$ formulas as atoms. As a minimal interpretation we have both $q' = \{\text{“}\Box A\text{”}\}$ and $q' = \{\text{“}\Box \mathit{false}\text{”}\}$. Since in both cases we have that $\partial(\psi, \epsilon) = \mathit{true}$, at the end of the iteration we have:

$$\begin{aligned} q_0 &= \{\Box A\} \\ Q &= \{q_0, \{\Box \mathit{false}\}, \emptyset\} \\ F &= \{q_0, \{\Box \mathit{false}\}, \emptyset\} \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset), \\ &\quad (q_0, \{A\}, q_0), (q_0, \{A\}, \{\Box \mathit{false}\})\} \end{aligned}$$

   - with $\Pi = \{\}$ we have

$$q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$$

$$\models \partial(\Box A, \Pi)$$

$$\models \partial(A, \Pi) \wedge \partial(\bullet\Box A, \Pi)$$

$$\models \mathit{false} \wedge (\text{“}\Box A\text{”} \vee \text{“}\Box \mathit{false}\text{”})$$

Which is always false. Thus we do not change nothing.

2. Iteration: we already analyzed $q = \{\Box A\}$, so we analyze $q = \{\Box \mathit{false}\}$
   - Both with $\Pi = \{\}$ and $\Pi = \{A\}$ we have that:

$$q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$$

$$\models \partial(\Box false, \Pi)$$

$$\models \partial(false, \Pi) \wedge \partial(\bullet\Box false, \Pi)$$

$$\models \mathit{false} \wedge (\text{“}\Box false\text{”} \vee \text{“}\Box \mathit{false}\text{”})$$

Which is always false. Thus we do not change nothing.

The NFA $\mathcal{A}_\varphi = \langle 2^{\{A\}}, Q, q_0, \delta, F\rangle$ is depicted in Figure 2.2, whereas the associated DFA is in Figure 2.3.

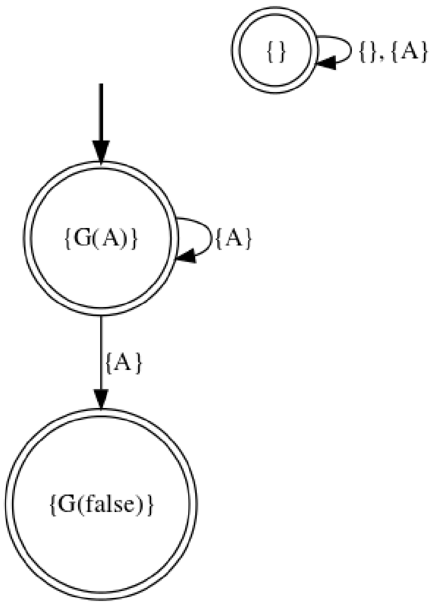


Figure 2.2. The NFA associated to $\Box A$. $G(A)$ stands for $\Box A$

Example 2.10. Analogously to what we did in 2.9, we see a run of the Algorithm 2.1, with the LTL$_f$ formula $\Diamond A$ (A atomic).

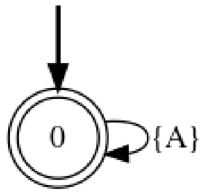


Figure 2.3. The DFA associated to $\Box A$

0. Set up:

$$\begin{aligned} q_0 &= \{\Diamond A\} \\ Q &= \{q_0, \emptyset\} \\ F &= \{\emptyset\} \qquad (\text{because } \partial(\Diamond A, \epsilon) = \partial(\mathit{true}\,\mathcal{U}\, A, \epsilon) = \mathit{false}) \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset)\} \end{aligned}$$

1. Iteration: analyze $q = \{\Diamond A\}$

   - with $\Pi = \{A\}$ we have

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Diamond A, \Pi) \\ &\models \partial(A, \Pi) \lor \partial(\circ \Diamond A, \Pi) \\ &\models \mathit{true} \lor (\text{“}\Diamond A\text{”} \land \text{“}\Diamond \mathit{true}\text{”}) \end{aligned}$$

   Since the propositional formula is trivially true, as a minimal interpretation we have $q' = \emptyset$. Considering that the empty conjunction is considered as *true* (as explained in Section 2.5), at the end of the iteration we have:

$$\begin{aligned} q_0 &= \{\Diamond A\} \\ Q &= \{q_0, \emptyset\} \\ F &= \{\emptyset\} \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset), (q_0, \{A\}, \emptyset)\} \end{aligned}$$

   - with $\Pi = \{\}$ we have

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Diamond A, \Pi) \end{aligned}$$

$$\models \partial(A,\Pi) \vee \partial(\circ\Diamond A,\Pi)$$

$$\models \mathit{false} \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”})$$

As a minimal interpretation we have $q' = \{\text{“}\Diamond A\text{”}, \text{“}\Diamond \mathit{true}\text{”}\}$. Since $\partial(\Diamond A,\epsilon) \wedge \partial(\Diamond \mathit{true},\epsilon) = \mathit{false} \wedge \mathit{false} \neq \mathit{true}$, we do not add q' to the accepting states F. Thus we have:

$$\begin{aligned}
q_0 &= \{\Diamond A\}\\
Q &= \{q_0, \emptyset, \{\Diamond A, \Diamond \mathit{true}\}\}\\
F &= \{\emptyset\}\\
\delta &= \{(\emptyset,\{\},\emptyset),(\emptyset,\{A\},\emptyset),\\
&\quad (q_0,\{A\},\emptyset),\\
&\quad (q_0,\{\},\{\Diamond A,\Diamond \mathit{true}\})\}
\end{aligned}$$

2. Iteration: we already analyzed $q = \{\Diamond A\}$, so we analyze $q = \{\Diamond A, \Diamond \mathit{true}\}$

- with $\Pi = \{\}$ we have that:

$$q' \models \bigwedge_{(\psi\in q)} \partial(\psi,\Pi)$$

$$\models \partial(\Diamond A,\Pi) \wedge \partial(\Diamond \mathit{true},\Pi)$$

$$\models [\partial(A,\Pi) \vee \partial(\circ\Diamond A,\Pi)] \wedge [\partial(\mathit{true},\Pi) \vee \partial(\circ\Diamond \mathit{true},\Pi)]$$

$$\models [\partial(A,\Pi) \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”})] \wedge [\mathit{true} \vee (\text{“}\Diamond \mathit{true}\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”})]$$

$$\models \partial(A,\Pi) \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”})$$

$$\models \mathit{false} \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”})$$

  As in the previous iteration, the minimal model is $q' = \{\text{“}\Diamond A\text{”}, \text{“}\Diamond \mathit{true}\text{”}\}$. Hence we add a new transition $(\{\Diamond A, \Diamond \mathit{true}\}, \{\}, \{\Diamond A, \Diamond \mathit{true}\})$.

- with $\Pi = \{A\}$ the delta-expansion is the same, except for the last step, where:

$$q' \models \mathit{true} \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”})$$

  The formula is always true, hence the minimal model is $q' = \emptyset$ and we add a new transition $(\{\Diamond A, \Diamond \mathit{true}\}, \{\}.\emptyset)$.

The NFA $\mathcal{A}_\varphi$ is then composed by:

$$\begin{aligned}
q_0 &= \{\Diamond A\}\\
Q &= \{q_0, \emptyset, \{\Diamond A, \Diamond \mathit{true}\}\}
\end{aligned}$$

$$\begin{aligned}F &= \{\emptyset\}\\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset),\\ &\quad (q_0, \{A\}, \emptyset),\\ &\quad (q_0, \{\}, \{\Diamond A, \Diamond true\})\\ &\quad (\{\Diamond A, \Diamond true\}, \{\}, \{\Diamond A, \Diamond true\})\\ &\quad (\{\Diamond A, \Diamond true\}, \{\}, \emptyset)\}\end{aligned}$$

The NFA $\mathcal{A}_\varphi = \langle 2^{\{A\}}, Q, q_0, \delta, F\rangle$ is depicted in Figure 2.4, whereas the associated DFA is in Figure 2.5.

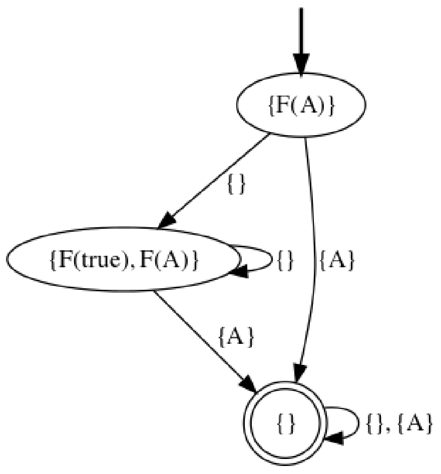


Figure 2.4. The NFA associated to $\Diamond A$. $F(A)$ stands for $\Diamond A$

Example 2.11. We list other examples of $\mathcal{A}_\varphi$ given a LTL$_f$/LDL$_f$ formula φ, obtained by Algorithm 2.1:

- *Conditional Response*: the LTL$_f$ formula $\varphi = \Box(A \Rightarrow \Diamond B)$ or equivalently the LDL$_f$ formula $\varphi = [true^*](\langle A\rangle tt \Rightarrow \langle true^*\rangle\langle B\rangle tt)$ translates into the automaton depicted in Figure 2.6.

- *Alternating sequence*: the LDL$_f$ formula $\varphi = \langle(A; B)^*\rangle End$ translates into the automaton depicted in Figure 2.7.

2.6 On-the-fly DFA

In this section, we describe a way to evaluate a LTL$_f$/LDL$_f$ formula without the need to build the entire automaton $\mathcal{A}_\varphi$. After that, we devise a new algorithm, a variant of LDL$_f$2NFA, that avoids the computation of the NFA, but directly translate the formula into a DFA. We provide some examples to clarify the presented topics.

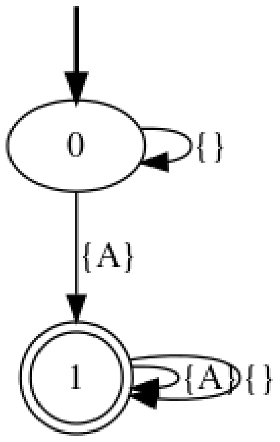


Figure 2.5. The DFA associated to $\Diamond A$

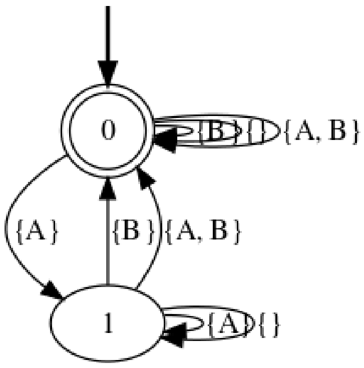


Figure 2.6. The DFA associated to $\varphi = \Box(A \Rightarrow \Diamond B)$

2.6.1 On-the-fly LTL$_f$/LDL$_f$ evaluation

In this section, we describe an alternative method to evaluate a trace on a DFA without the need for constructing $\mathcal{A}_\varphi$, that we call *on-the-fly* (Brafman et al., 2018). The idea is, we progress all possible states that the NFA can be in, after consuming the next trace symbol, and accept the trace iff, once it has been completely read, the set of possible states contains a final state.

More formally, call a set of possible states for the NFA a macrostate, let $Q = \{q_1, \dots, q_n\}$ be the current macrostate (initially $Q = Q_0 = \{q_0\} = \{\{\varphi\}\}$), and let Π be the next trace symbol. Then, the successor macrostate is the set Q' defined as

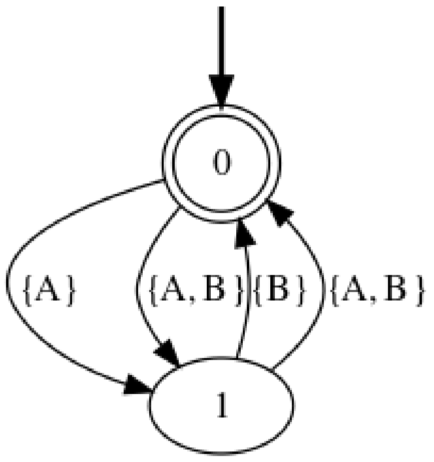


Figure 2.7. The DFA associated to $\varphi = \langle (A; B)^* \rangle End$

follows:

$$Q' = \{q' | \exists q \in Q \ s.t. \ q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)\} \tag{2.19}$$

Notice that the condition $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$ is the same of the one in line 12 of Algorithm 2.1. Given an input trace π, we decide whether $\pi \models \varphi$ by iterating the above procedure, starting from the initial state $Q = Q_0$, and accepting π iff the last state obtained includes $\{true\}$, considering their evaluation in the empty trace (i.e. with $\partial(\psi, \epsilon)$). We denote the evaluation over the empty trace of a macrostate $Q = \{q_1, \dots, q_n\}$ as Q^ϵ. Formally:

$$Q^\epsilon = \{\varphi | \varphi = \bigwedge_{\psi \in q_i} \partial(\psi, \epsilon)\} \tag{2.20}$$

Example 2.12. Consider Example 2.9 with $\varphi = \Box A$, we show how the on-the-fly evaluation of traces works. At the beginning, we have that

$$Q = Q_0 = \{\{\Box A\}\}$$

In this example, we ask the following questions:

1. $\langle\rangle \models \varphi$? Does the empty trace $\pi = \langle\rangle$ satisfy the formula φ? We expect that the answer is yes, due to the semantics of $\Box A$. With the on-the-fly approach, we need to compute for each NFA state $q \in Q$ the conjunction between every $\partial(\psi, \epsilon)$, where $\psi \in q$. As said before, we consider the empty conjunction as $\{true\}$.

   In our example, the computation gives us:

   $$Q_0^\epsilon = \{\{true\}\}$$

   because $\partial(\Box A, \epsilon) = true$. Since Q_0^ϵ contains $\{true\}$, Q_0^ϵ is an accepting state, hence $\pi \models \Box A$, as expected.

2. $\langle\{\}\rangle \models \varphi$? This time consider the trace $\pi = \langle\{\}\rangle$ or, equivalently, $\pi = \langle\neg A\rangle$. We expect that the on-the-fly evaluation returns *false*, hence $\pi \not\models \varphi$. In order to answer, we need to progress the automaton on-the-fly for each element of the trace (in this case only one) and check if the last macrostate is an accepting state by using the procedure explained in the previous case. The next macrostate Q_1 by applying Equation 2.19. Actually, it is computed as we did in Iteration 1 of Example 2.9 with $\Pi = \{\}$.

   Since no q' can be found, $Q_1 = \{\}$, which is not an accepting state since $\{true\} \notin Q_1^\epsilon$. Hence, $\pi \not\models \varphi$.

3. $\langle\{A\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{\}\rangle$. We expect that the on-the-fly evaluation returns *true*, hence $\pi \models \varphi$. We proceed, as in the previous case, to compute the next macrostate Q_1 by applying Equation 2.19. Actually, it is computed as we did in Iteration 1 with $\Pi = \{A\}$. As a minimal interpretation we have both $q' = \{\Box A\}$ and $q' = \{\Box \mathit{false}\}$. Hence, the new macrostate is $Q_1 = \{\{\Box A\}, \{\Box \mathit{false}\}\}$.

   Since there are no other symbols in the trace π to be processed, we compute $Q_1^\epsilon = \{\{true\}, \{true\}\} = \{\{true\}\}$. Since $\{true\} \in Q_1^\epsilon$, we can say that $\pi \models \varphi$.

4. $\langle\{A\}, \{\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{A\}, \{\}\rangle$. We expect that the on-the-fly evaluation returns *false*, hence $\pi \not\models \varphi$. We proceed, as in the previous case, to compute the next macrostates by applying Equation 2.19. Macrostate Q_1 is the same as we have seen in Case 3. We apply again the progression rule of Equation 2.19 with $\Pi = \pi(2) = \{\}$. As a minimal interpretation we have both $q' = \{\Box A\}$ and $q' = \{\Box \mathit{false}\}$. Hence, the new macrostate is $Q_2 = \{\}$. as we've seen in Iteration 2 of Example 2.9.

   Since there are no other symbols in the trace π to be processed, we compute $Q_2^\epsilon = \{\}$. Since $\{true\} \notin Q_2^\epsilon$, we can say that $\pi \not\models \varphi$.

5. $\langle\{\}, \{A\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{\}, \{A\}\rangle$. We expect that the on-the-fly evaluation returns *false*, hence $\pi \not\models \varphi$. The macrostate Q_1 is the same as we have seen in Case 2, i.e. $Q_1 = \{\}$. We apply again the progression rule of Equation 2.19 with $\Pi = \pi(2) = \{A\}$. But this is trivially $Q_2 = \{\}$, by definition of the progression rule.

   Since there are no other symbols in the trace π to be processed, we compute $Q_2^\epsilon = \{\}$. Since $\{true\} \notin Q_2^\epsilon$, we can say that $\pi \not\models \varphi$.

Notice how we use the same progression of Algorithm 2.1, but instead of aiming to build the entire automaton, we focus only on the states that are relevant for the satisfaction of the formula, given a trace.

Example 2.13. Analogously as we did in Example 2.12 for Example 2.9, we consider Example 2.10 with $\varphi = \Diamond A$, and we show how the on-the-fly evaluation of traces works also in this case. At the beginning, we have that

$$Q = Q_0 = \{\{\Diamond A\}\}$$

In this example, we ask the following questions:

1. $\langle\rangle \models \varphi$? Does the empty trace $\pi = \langle\rangle$ satisfy the formula φ? We expect that the answer is no, due to the semantics of $\Diamond A$.

   We observe that $Q_0^\epsilon = \{\}$, because $\partial(\Diamond A, \epsilon) = \mathit{false}$.

   Since Q_0^ϵ does not contain $\{\mathit{true}\}$, Q_0^ϵ is not an accepting state, hence $\pi \not\models \Diamond A$, as expected.

2. $\langle\{\}\rangle \models \varphi$? This time consider the trace $\pi = \langle\{\}\rangle$ or, equivalently, $\pi = \langle\neg A\rangle$. We expect that the on-the-fly evaluation returns *false*, hence $\pi \not\models \varphi$. In order to answer, we need to progress the automaton on-the-fly for each element of the trace (in this case only one) and check if the last macrostate is an accepting state by using the procedure explained in the previous case. The next macrostate Q_1 by applying Equation 2.19. Actually, it is computed as we did in Iteration 1 of Example 2.10 with $\Pi = \{\}$. As a minimal interpretation we have $q' = \{\Diamond A, \Diamond \mathit{true}\}$. Hence, the new macrostate is $Q_1 = \{\{\Diamond A, \Diamond \mathit{true}\}\}$.

   Now, $Q_1^\epsilon = \{\{\mathit{false}\}\}$, which is not an accepting state since $\{\mathit{true}\} \notin Q_1^\epsilon$. Hence, $\pi \not\models \varphi$.

3. $\langle\{A\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{\}\rangle$. We expect that the on-the-fly evaluation returns *true*, hence $\pi \models \varphi$. We proceed, as in the previous case, to compute the next macrostate Q_1 by applying Equation 2.19. Actually, it is computed as we did in Iteration 1 with $\Pi = \{A\}$. As a minimal interpretation we have $q' =$. Hence, the new macrostate is $Q_1 = \{\emptyset\}$.

   Since there are no other symbols in the trace π to be processed, we compute $Q_1^\epsilon = \{\{\mathit{true}\}\}$. Since $\{\mathit{true}\} \in Q_1^\epsilon$, we can say that $\pi \models \varphi$.

4. $\langle\{A\}, \{\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{A\}, \{\}\rangle$. We expect that the on-the-fly evaluation returns *true*, and so $\pi \models \varphi$. We proceed, as in the previous case, to compute the next macrostates by applying Equation 2.19. Macrostate Q_1 is the same as we have seen in Case 3. We apply again the progression rule of Equation 2.19 with $\Pi = \pi(2) = \{\}$, that leads to the new macrostate is $Q_2 = \{\emptyset\}$. Notice that $Q_1 = Q_2$.

   Since there are no other symbols in the trace π to be processed, we compute $Q_2^\epsilon = \{\{\mathit{true}\}\}$. Since $\{\mathit{true}\} \in Q_2^\epsilon$, we can say that $\pi \models \varphi$.

5. $\langle\{\}, \{A\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{\}, \{A\}\rangle$. We expect that the on-the-fly evaluation returns *true*, hence $\pi \models \varphi$. The macrostate Q_1 is the same as we have seen in Case 2, i.e. $Q_1 = \{\{\Diamond A, \Diamond \mathit{true}\}\}$. We apply again the progression rule of Equation 2.19 with $\Pi = \pi(2) = \{A\}$. But this is trivially $Q_2 = \{\emptyset\}$ (as we've seen in Iteration 2 of Example 2.10), by definition of the progression rule.

   Since there are no other symbols in the trace π to be processed, we compute $Q_2^\epsilon = \{\{\mathit{true}\}\}$. Since $\{\mathit{true}\} \in Q_2^\epsilon$, we can say that $\pi \models \varphi$.

Notice how we use the same progression of Algorithm 2.1, but instead of aiming to build the entire automaton, we focus only on the states that are relevant for the satisfaction of the formula, given a trace.

2.6.2 LDL$_f$2DFA: a variant of LDL$_f$2NFA

Example 2.12 and 2.13 suggest a new way to translate LTL$_f$/LDL$_f$ formulas to automata, that is a variant of LDL$_f$2NFA (Algorithm 2.1). We call it LDL$_f$2DFA, and directly translates a LTL$_f$/LDL$_f$ formula to a DFA, instead of first translation into a NFA and then compute the DFA by determinization.

Algorithm 2.2. LDL$_f$2DFA: from LTL$_f$/LDL$_f$ formula φ to DFA $\mathcal{A}_\varphi$

```
1: input LDL_f/LTL_f formula $\varphi$
2: output DFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, \mathcal{Q}, Q_0, \delta, F \rangle$     ▷ Notice: $\mathcal{Q}$ is a set of macrostates.
3: $Q_0 \leftarrow \{\{\varphi\}\}$     ▷ the initial state of $\mathcal{A}_\varphi$ is the initial macrostate
4: $F \leftarrow \emptyset$
5: $\mathcal{Q} \leftarrow \{Q_0\}$
6: $\delta \leftarrow \emptyset$
7: if $(\{true\} \in Q_0^\epsilon)$ then
8:     $F \leftarrow F \cup \{Q_0\}$
9: end if
10: while ($\mathcal{Q}$ or $\delta$ change) do
11:     for $(Q \in \mathcal{Q}, \Pi \in 2^{\mathcal{P}})$ do
12:         $Q' \leftarrow \{\}$
13:         for $(q \in Q)$ do     ▷ Conceptually, the same loop of Algorithm 2.1, line 11
14:             if $(q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi))$ then
15:                 $Q' \leftarrow Q' \cup \{q'\}$
16:             end if
17:         end for
18:         $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{Q'\}$     ▷ Add new macrostate $Q$ to the set of macrostates $\mathcal{Q}$
19:         $\delta \leftarrow \delta \cup \{(Q, \Pi, Q')\}$
20:         if $(\{true\} \in Q'^\epsilon)$ then
21:             $F \leftarrow F \cup \{Q'\}$
22:         end if
23:     end for
24: end while
```

The idea behind Algorithm 2.2 is the following: build the DFA by doing the exploration of automaton states and determinization *at the same time*. Indeed, each macrostate tracks all the possible paths (according to the trace symbols processed) of the "implicit" NFA. The computation of the next NFA state, i.e. the for-loop at line 13, works in the same way of the for-loop in line 11 of Algorithm 2.1. For a single macrostate Q, given a propositional interpretation Π, the operation is made for every NFA state $q \in Q$. The next macrostate Q' is then composed by all the next NFA states q'. Given the triple Q, Π, Q', we actually have a transition of the DFA $\mathcal{A}_\varphi$. Doing this operation for every macrostate and for every interpretation, until convergence, will eventually lead to the exploration of every macrostate and transitions among them.

The main advantage over Algorithm 2.1 is that we avoid the state explosion due to the determinization procedure since we only process reachable states of the final DFA. We use this algorithm in the implementations of Chapter 3.

Example 2.14. We consider Example 2.9 but using Algorithm 2.2 for translation of

$\varphi = \Box A$ into a DFA.

0. Before the main loop, we have:

$$\begin{aligned} Q_0 &= \{\{\Box A\}\} \\ \mathcal{Q} &= \{Q_0\} \\ \delta &= \emptyset \\ F &= \{Q_0\} \quad \text{(because } \{\mathit{true}\} \in Q_0^\epsilon) \end{aligned}$$

   The DFA at this stage is depicted in Figure 2.8a.

1. Iteration: Consider the macrostate Q_0. Consider the (unique) NFA state $\{\Box A\}$.

   With $\Pi = \{A\}$ we generate the new macrostate $Q' = \{\{\Box A\}, \{\Box \mathit{false}\}\}$. We add Q' to $\mathcal{Q}$ and (Q_0, Π, Q') to δ. We followed the same steps as we did in Example 2.12, Case 3.

   With $\Pi = \{\}$ we generate the new macrostate $Q' = \emptyset$. We add Q' to $\mathcal{Q}$ and (Q_0, Π, Q') to δ. Since $\{\mathit{true}\} \notin \mathcal{Q}'^\epsilon$, we do not add Q' to F. We followed the same steps as we did in Example 2.12, Case 2.

   The DFA at this stage is depicted in Figure 2.8b.

2. Iteration: We already processed $Q = \{\{\Box A\}\}$.

   Consider the macrostate $Q = \emptyset$. Since there exists no $q \in Q$, the for-loop at line 13, we add to δ all the transitions of the form $(\emptyset, \Pi, \emptyset)$, for all $\Pi \in 2^{\mathcal{P}}$.

   Now consider the macrostate $Q = \{\{\Box A\}, \{\Box \mathit{false}\}\}$.

   Consider $\Pi = \{A\}$. For $q = \{\Box A\}$ we generate the sub-macrostate $q' = \{\Box A\}$ and $q' = \{\Box \mathit{false}\}$. For $q = \{\Box \mathit{false}\}$ we do not generate any sub-macrostate. Hence, the resulting macrostate is $Q' = \{\{\Box A\}, \{\Box \mathit{false}\}\}$. Since $Q' \in \mathcal{Q}$, we only add (Q, Π, Q') to δ.

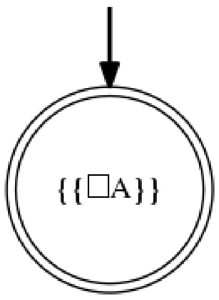
   Consider $\Pi = \{\}$. For $q = \{\Box A\}$ we do not generate any sub-macrostate. For $q = \{\Box \mathit{false}\}$ we do not generate any sub-macrostate. Hence, the resulting macrostate is $Q' = \{\}$. Since $Q' \in \mathcal{Q}$, we only add (Q, Π, Q') to δ.

   Since there are no other macrostates nor propositional interpretation to process, the algorithm terminates. The final DFA is depicted in Figure 2.8c.

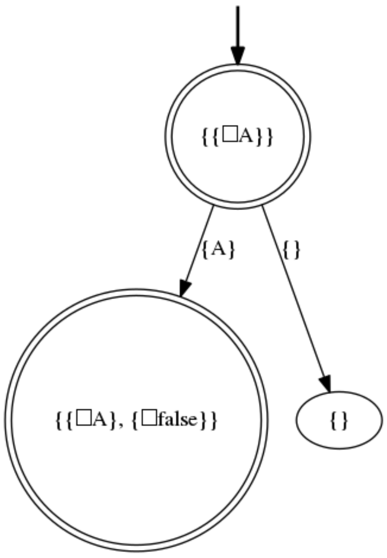
It is interesting to observe that the macrostate $Q = \{\{\Box A\}, \{\Box \mathit{false}\}\}$, where we end after reading the symbol $\{A\}$ from the initial state, is the set of NFA states (Figure 2.2) where we could ends after reading the same symbol from the initial state, namely $\{\Box A\}$ and $\{\Box \mathit{false}\}$. Analogous considerations can be made for other symbols and other macrostates.

This observation makes clearer the true meaning of Algorithm 2.2 with respect to Algorithm 2.1: that is, the macrostates keep track of all the possible evolutions of the NFA with respect to a trace π. By reading all the possible symbols from every macrostate, we will eventually discover all the relevant states of the DFA.

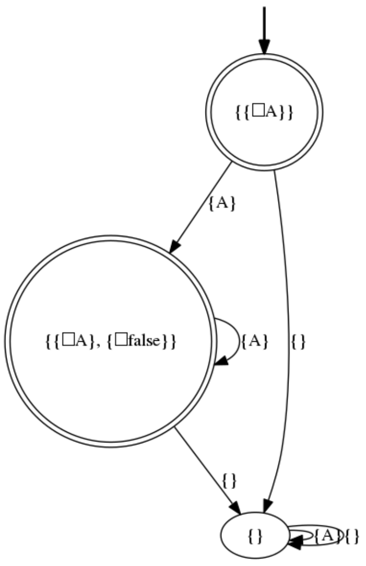
Furthermore, the minimization and trimming of the resulting automaton (shown in Figure 2.8c) yield the one shown in Figure 2.3, as an evidence of the equivalence between the two algorithms.



(a) Iteration 0



(b) Iteration 1



(c) Iteration 2

Figure 2.8. The automaton of Example 2.12, step by step.

Example 2.15. We consider Example 2.10 but using Algorithm 2.2 for translation of $\varphi = \Diamond A$ into a DFA.

0. Before the main loop, we have:

$$\begin{aligned} Q_0 &= \{\{\Diamond A\}\} \\ \mathcal{Q} &= \{Q_0, \emptyset\} \\ \delta &= \emptyset \\ F &= \emptyset \quad \text{(because } \{\mathit{true}\} \notin Q_0^\epsilon) \end{aligned}$$

   The DFA at this stage is depicted in Figure 2.9a.

1. Iteration: Consider the macrostate Q_0. Consider the (unique) NFA state $\{\Diamond A\}$.

   With $\Pi = \{A\}$ we generate the new macrostate $Q' = \{\emptyset\}$. We add Q' to $\mathcal{Q}$ and (Q_0, Π, Q') to δ. Since $\{\mathit{true}\} \in \mathcal{Q}'^\epsilon$, we add Q' to F. We followed the same steps as we did in Example 2.13, Case 3.

   With $\Pi = \{\}$ we generate the new macrostate $Q' = \{\{\Diamond A, \Diamond \mathit{true}\}\}$. We add Q' to $\mathcal{Q}$ and (Q_0, Π, Q') to δ. Since $\{\mathit{true}\} \notin \mathcal{Q}'^\epsilon$, we do not add Q' to F. We followed the same steps as we did in Example 2.13, Case 2.

   The DFA at this stage is depicted in Figure 2.9b.

2. Iteration: We already processed $Q = \{\{\Box A\}\}$.

   Consider the macrostate $Q = \{\emptyset\}$. Since the unique successor state of $q = \emptyset$ is $q' = \emptyset$, the next macrostate, for every symbol, is the same. Hence, we add to δ all the transitions of the form $(\{\emptyset\}, \Pi, \{\emptyset\})$, for all $\Pi \in 2^{\mathcal{P}}$.

   Now consider the macrostate $Q = \{\{\Diamond A, \Diamond \mathit{true}\}\}$.

   Consider $\Pi = \{A\}$. As we've seen in Case 5 of Example 2.13, the next macrostate is $Q' = \{\emptyset\}$, since that for $q = \{\Diamond A, \Diamond \mathit{true}\}$ the successor state is $q' = \emptyset$. Since $Q' \in \mathcal{Q}$, we only add (Q, Π, Q') to δ.

   Consider $\Pi = \{\}$. The successor state of $q = \{\Diamond A, \Diamond \mathit{true}\}$ is $q' = q$. Hence, the resulting macrostate is $Q' = \{\{\Diamond A, \Diamond \mathit{true}\}\}$. Since $Q' \in \mathcal{Q}$, we only add (Q, Π, Q') to δ. We followed the same steps as we did in Example 2.10, Iteration 2.

   Since there are no other macrostates nor propositional interpretation to process, the algorithm terminates. The final DFA is depicted in Figure 2.9c.

2.7 Reasoning in LTL$_f$/LDL$_f$

In this section, we study the complexity of LTL$_f$/LDL$_f$ reasoning (i.e. complexity of problems as defined in Definition 2.5, by leveraging the automata construction described in previous sections.

Theorem 2.7 (De Giacomo and Vardi (2013)). *Satisfiability, validity, and logical implication for LDL$_f$ formulas are PSPACE-complete*

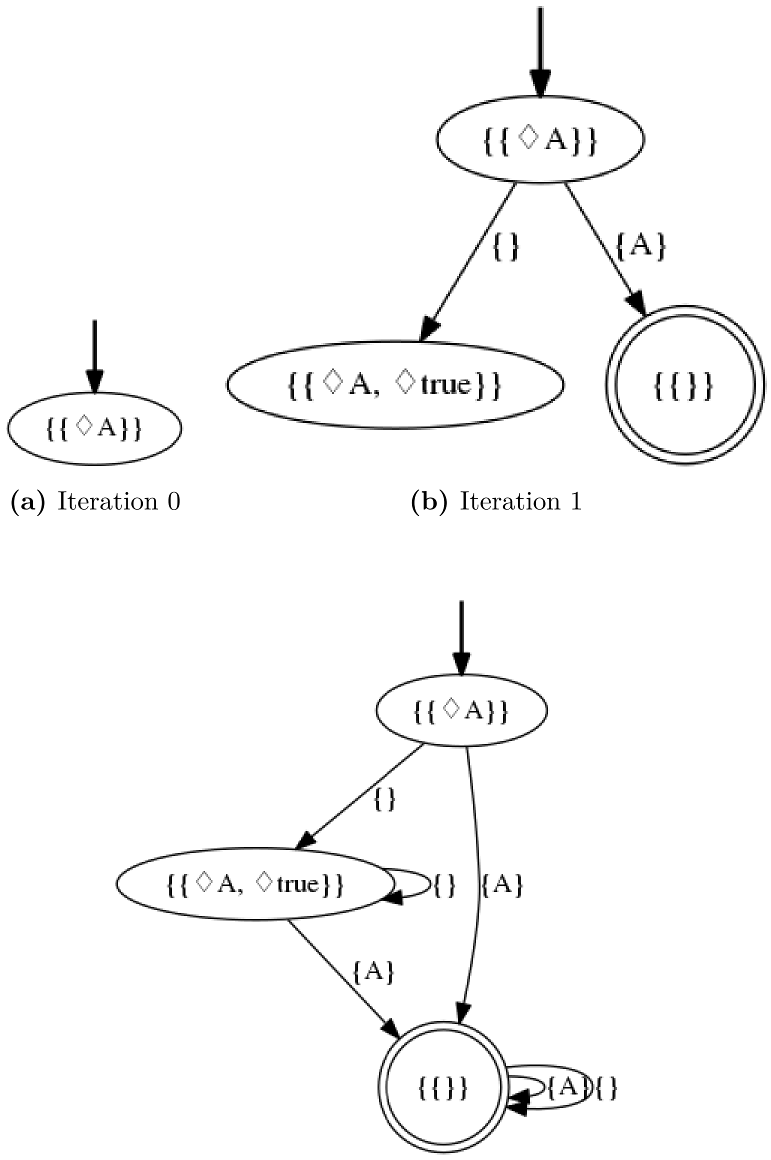
Proof. Given a LTL$_f$/LDL$_f$ φ, we can leverage Theorem 2.6 to solve these problems, namely:

- For LTL$_f$/LDL$_f$ satisfiability we compute the associated NFA (as explained in Section 2.5 (which is an exponential step) and then check NFA for nonemptiness (NLOGSPACE).
- For LTL$_f$/LDL$_f$ validity we compute the NFA associated to $\neg\varphi$ (which is an exponential step) and then check NFA for nonemptiness (NLOGSPACE).
- For LTL$_f$/LDL$_f$ logical implication $\psi \models \varphi$ we compute the NFA associated to $\psi \wedge \neg\varphi$ (which is an exponential step) and then check NFA for nonemptiness (NLOGSPACE).

□

2.8 Conclusion

In this chapter, we provided the logical tools to face other topics in later chapters. We introduced several formal languages that allowed us to introduce LTL$_f$ and LDL$_f$, focusing on their interesting properties. Moreover, we described in detail the procedure for translation from LTL$_f$/LDL$_f$ formulas to DFAs, which yields an effective way of reasoning about LTL$_f$/LDL$_f$ formulas.



(a) Iteration 0 **(b)** Iteration 1

(c) Iteration 2

Figure 2.9. The automaton of Example 2.13, step by step.

Chapter 3

FLLOAT

In this chapter, we describe FLLOAT (From LTL$_f$/LDL$_f$ tO AutomaTa), a software project written in Python. It is a porting of the homonym software project written in Java. It is the implementation of many of the topics described in Chapter 2.

3.1 Introduction

Main features: FLLOAT is a Python library that provides support for:

- Syntax, semantics and parsing of the following logic formalisms:
  - Propositional Logic;
  - Linear Temporal Logic on Finite Traces LTL$_f$
  - Linear Dynamic Logic on Finite Traces LDL$_f$;
- Conversion from LTL$_f$/LDL$_f$ formula to NFA, DFA and DFA On-The-Fly

Dependencies: FLLOAT requires Python>=3.5 and depends on the following packages:

- PLY, a pure-Python implementation of the popular compiler construction tools Lex and Yacc. It has been used for the parsing of PL and LTL$_f$/LDL$_f$ formulas;
- Pythomata, a Python package which provides support for NFA, DFA, determinization and minimization algorithms and reasoning on DFAs. It has been used for deal with $\mathcal{A}_\varphi$, the equivalent automaton of a LTL$_f$/LDL$_f$ formula.

Installation: You can find the package on PyPI, hence you can install it with:

```
pip install flloat
```

Please go here for further details.

The software is open source and is released under MIT license.

3.2 Package structure

The package is structured as follows:

- `flloat.py`: the main module, it contains the implementation of the translation from LTL$_f$/LDL$_f$ formulas to automata. The functions implemented here are called from methods of LTL$_f$/LDL$_f$ formulas.

- `base/`: contains the abstract definitions used in other modules. The main modules are:

  - `Symbol.py` and `Symbols.py`, where have been defined the class `Symbol` to represent the atomic propositional symbols and the operator symbols;
  - `Alphabet.py`, which is an abstraction for manage a set of `Symbol`;
  - `Interpretation.py`, an abstract class denoting the semantics used for truth evaluation. E.g. for PL the corresponding interpretation is `PLInterpretation` (a set of `Symbol`), whereas for LTL$_f$/LDL$_f$ we have `FiniteTrace`, which is a list of `PLInterpretation`.
  - `Formula.py`, the module containing the base class `Formula`. Every formula class extends `Formula`. In this module are defined also `AtomicFormula`, `Operator`, `BinaryOperator` etc., and how to build a syntax tree.
  - `truths.py` and `nnf.py` that provide abstract implementations for truth evaluations of formulas and negation normal form operations.
  - other abstractions definitions that are implemented for each extending subclass.

- `syntax/`: modules for each formalism (i.e. `pl.py`, `ltlf.py` and `ldlf.py`). In those modules are declared all the classes for representing formulas, implementing their truth evaluation procedure taking into account their correlation (e.g. `And` is the negative dual of `Or`, you can define `Implies` in terms of `Not` and `Or` etc.);

- `semantics/`: modules providing implementations for the semantics. E.g. you can find `PLInterpretation` and `FiniteTrace` cited before;

- `parser/`: modules where are defined the parsers of formulas in PL and LTL$_f$/LDL$_f$. They depends on PLY.

In the following sections we show typical use cases, describe the used APIs and look at their implementation.

3.3 Propositional Logic formulas

FLLOAT provides support for Propositional Logic (PL). In the following, we will see both examples and explanation of the code.

3.3.1 Specify a PL formula

The easiest way to specify a propositional formula is to use a `PLParser`:

```python
from flloat.parser.pl import PLParser

parser = PLParser()
```

For instance, the propositional formula $\varphi = A \wedge (\neg B \vee C) \wedge D$ can be represented in this way:

```python
phi = parser("A␣&␣(!B␣|␣C)␣&␣D")
```

Which generate the syntax tree in Figure 3.1. The logical operators symbols $\neg, \wedge, \vee$ are replaced by, respectively, `!`, `&`, `|`. In red are represented the symbols used to build the formula, for which the associated class is `Symbol`. On the other hand, in blue are shown the classes used to build the syntax tree, that represent a particular formula/operator in PL. They are: `PLAtomic`, used to represent atomic propositions; `PLNot`, `PLAnd` and `PLOr` for the associated logic operators. All of them extends from the base class `Formula`.

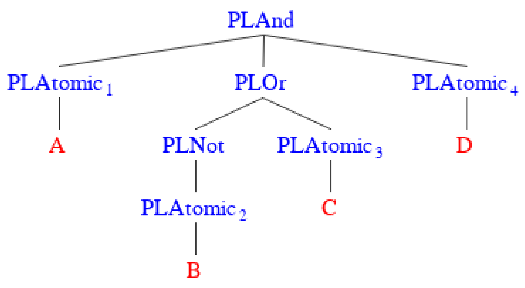


Figure 3.1. The syntax tree generated by the parsing of `"A & (!B | C) & D"`, In red are represented instances of the class `Symbol` , whereas in blue the instances of the formulas/operators classes used to build the syntax tree.

We could construct the same syntax structure by using other APIs of FLLOAT:

Listing 3.1. A "manually" constructed PL formula.

```python
from flloat.base.Symbol import Symbol
from flloat.syntax.pl import PLAtomic, PLAnd, PLOr, PLNot

A, B = Symbol("A"), Symbol("B")
C, D = Symbol("C"), Symbol("D")

atomic_A = PLAtomic(A)
atomic_B = PLAtomic(B)
atomic_C = PLAtomic(C)
atomic_D = PLAtomic(D)

not_B = PLNot(atomic_B)
B_or_C = PLOr([not_B, atomic_C])
phi = PLAnd([atomic_A, B_or_C, atomic_D])
```

Now follow the implementation details about how the above code can run.

The Symbol class

Symbol is one of the base classes, declared in Symbol.py, whose code is reported in Listing 3.2.

Listing 3.2. The Symbol class.

```python
class Symbol(str, Hashable):
  def __init__(self, name:str):
    str.__init__(self)
    Hashable.__init__(self)
    self.name = name

  def _members(self):
    return self.name
```

The class Symbol can be thought as a wrapper for the type str. Additionally, it inherits some useful properties from the class Hashable, implemented for performance purposes, which is reported in Listing 3.3:

Listing 3.3. The Hashable class.

```python
from abc import ABC, abstractmethod
from copy import copy

class Hashable(ABC):

  def __init__(self):
    self._hash = None

  @abstractmethod
  def _members(self):
    raise NotImplementedError

  def __eq__(self, other):
    if type(other) is type(self):
      return self._members() == other._members()
    else:
      return False

  def __hash__(self):
    if self._hash is None:
      self._hash = hash(self._members())
    return self._hash

  def __getstate__(self):
    d = copy(self.__dict__)
    d.pop("_hash")
    return d

  def __setstate__(self, state):
    self.__dict__ = state
    self._hash = None
```

Its main purpose is to allow memoization of the hash value. This is useful for immutable objects since their hash value does not change during their life cycle. Furthermore, the mathematical domain of interest requires hash-based data structure such as sets and dictionaries, so it is a good idea to compute the hash only once and retrieve the precomputed value when needed. The `Hashable` class is inherited from many other classes since the immutable property is satisfied by many entities. It requires that all the classes that inherit from it implement a `__members` function, which returns a hashable signature of the object that identifies it uniquely. The methods `__getstate__` and `__setstate__` simply do not include the precomputed hash value, since it might have a different value among different Python processes, due to hash randomization.

The `Formula` abstract class and other abstraction

In `flloat/base/Formula.py` there are many abstraction that allow the construction of the syntax tree. The three dots `...` represent code omissions.

Listing 3.4. `flloat/base/Formula.py` module

```python
from abc import abstractmethod, ABC
from typing import Sequence, Set

from flloat.base.Symbol import Symbol
from flloat.base.Symbols import Symbols
from flloat.base.hashable import Hashable


class Formula(Hashable):
  def __init__(self):
    super().__init__()

    ...

class AtomicFormula(Formula):
  def __init__(self, s:Symbol):
    super().__init__()
    self.s = s

  def _members(self):
    return self.s

  ...

class Operator(Formula):
  @property
  def operator_symbol(self) -> str:
    raise NotImplementedError

    ...

class UnaryOperator(Operator):
  def __init__(self, f: Formula):
    super().__init__()
    self.f = f.simplify()

  def _members(self):
    return (self.operator_symbol, self.f)

  ...

class BinaryOperator(Operator):
  def __init__(self, formulas:OperatorChilds):
    super().__init__()
    assert len(formulas) >= 2
    self.formulas = tuple(formulas)
    self.formulas = self._popup()

  ...

  def _members(self):
    return (self.operator_symbol, self.formulas)

  def _popup(self):
    """recursively find the same binary operator among
    child formulas and pop up them at the same level"""
    res = ()
    for child in self.formulas:
      if type(child) == type(self):
        superchilds = child._popup()
        res += superchilds
      else:
        res += (child, )
    return tuple(res)

class CommutativeBinaryOperator(BinaryOperator):
  def __init__(self, formulas:OperatorChilds):
    # Assuming idempotence: e.g. A & A === A
    super().__init__(formulas)
    # order does not matter -> set operation
    # remove duplicates -> set operation
    self.formulas_set = frozenset(self.formulas)
    # unique representation -> sorting
    self.members = tuple(
      sorted(self.formulas_set, key=lambda x: hash(x))
    )

```

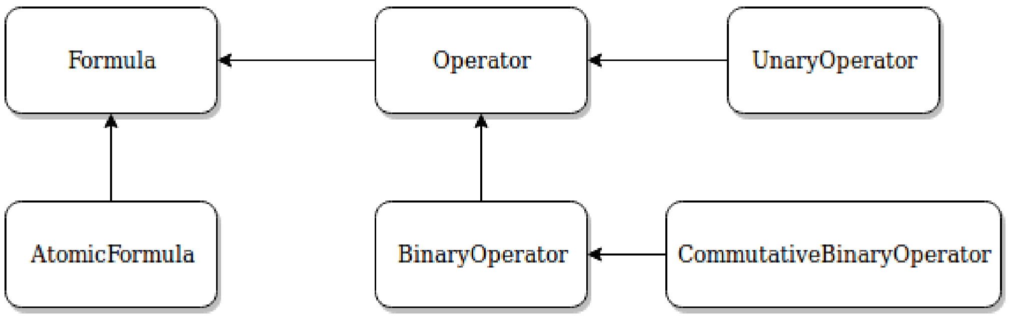


Figure 3.2. An inheritance schema for the classes in `flloat/base/Formula.py`. Arrows stand for "extends".

```
    ...
```

The class `Formula` is "the mother" of all the formulas. `AtomicFormula` represents a formula without any child in the syntax tree; it holds a `Symbol` object. `Operator` is an abstract class that represents a logical operator over formulas. It can be a `UnaryOperator` (e.g. the *not* $\neg$), a `BinaryOperator` (e.g. the *implies* $\Rightarrow$) and a `CommutativeBinaryOperator`, which is a binary operator that satisfies the commutative property (e.g. the *and* $\wedge$ and the *or* $\vee$). Every `Operator` has to have an operator symbol used for the conversion into strings.

When constructing a `BinaryOperator` instance, the list of subformulas is analyzed by looking for a subformula of the same type of the instance. This is a way to avoid multiple representations of the same syntax tree. For build a `BinaryOperator` formula, we need a *list* of subformulas, whereas for a `CommutativeBinaryOperator` we hold subformulas as a *set*, because order does not matter. However, for the latter case, in order to have a unique representation, we hold also a list of subformulas sorted by their hash.

We remark that,despite the name, `BinaryOperator` constructor take as input a *list* of formulas and not a *pair* of formulas. The reason, as stated before, is to reduce the depth of the syntax tree when the same binary operator is applied in chain many times.

Figure 3.2 depicts a schema to represent the relations between the just described abstractions.

`PLFormulas`: `PLAtomic`, `PLNot`, `PLAnd`, `PLOr` classes

The `PLAtomic`, `PLNot`, `PLAnd`, `PLOr` classes, as well as other PL classes for syntax support, are all declared in `flloat/syntax/pl.py` (Listing 3.5).

Listing 3.5. The `flloat/syntax/pl.py` module

```python
class PLTruth(Truth):
  @abstractmethod
  def truth(self, i: PLInterpretation, *args):
    raise NotImplementedError

class PLFormula(Formula, PLTruth, NNF):
  def __init__(self):
    Formula.__init__(self)

  def all_models(self, alphabet: _Alphabet):
  """Find all the possible interpretations
  given a set of symbols"""
    ...

  def minimal_models(self, alphabet: _Alphabet):
  """Find models of minimum size"""
    ...

class PLBinaryOperator(BinaryOperator, PLFormula):
  ...

class PLCommBinaryOperator(DualCommutativeOperatorNNF,
  PLFormula):
  ...

class PLAtomic(AtomicFormula, PLFormula):

  def truth(self, i:PLInterpretation, *args):
    return self.s in i

  def _to_nnf(self):
    return self

 def negate(self):
    return PLNot(self)

class PLTrue(PLAtomic):
  def __init__(self):
    PLAtomic.__init__(self, Symbol(Symbols.TRUE.value))

  def truth(self, *args):
    return True

  def negate(self):
    return PLFalse()


class PLFalse(PLAtomic):
  def __init__(self):
    PLAtomic.__init__(self, Symbol(Symbols.FALSE.value))

  def truth(self, *args):
    return False

  def negate(self):
    return PLTrue()
```

```

class PLNot(NotTruth, PLFormula, NotNNF):
  pass

class PLOr(PLCommBinaryOperator, OrTruth):
  pass

class PLAnd(PLCommBinaryOperator, AndTruth):
  pass

class PLImplies(PLBinaryOperator, ImpliesConvertible):
  And = PLAnd
  Or = PLOr
  Not = PLNot

class PLEquivalence(PLCommBinaryOperator,
  EquivalenceConvertible):
  And = PLAnd
  Or = PLOr
  Not = PLNot

PLOr.Dual = PLAnd
PLAnd.Dual = PLOr
```

Every propositional formula extends from `PLFormula`. It implements the methods `all_models` that, given an alphabet, returns all the propositional interpretations that satisfy the formula. `minimal_models` returns only the minimal ones. `PLBinaryOperator` and `PLCommBinaryOperator` are conceptually the same of `BinaryOperator` and `CommutativeBinaryOperator` described before.

Other classes define the possible PL formulas: `PLAtomic` is an `AtomicFormula`, hence its instances just hold a `Symbol` object; `PLTrue` and `PLFalse` are `PLAtomic` formulas with a fixed symbol that represent, respectively, propositional *true* and propositional *false*; `PLNot`, `PLAnd`, `PLOr` has been already described; `PLImplies` and `PLEquivalence` correspond to $\Rightarrow$ and $\Leftrightarrow$ logic operator, respectively.

It is interesting to observe that the implementation of some features such as syntax, semantics and conversion to Negation Normal Form (NNF) is inherited from other classes/interfaces, such as `Truth` and `NNF`. The implementation is made generically, i.e. by not assuming that we are talking about PL formulas. Indeed, the implementation of the same formulas (and, or, not etc.) is used for PL and LTL$_f$/LDL$_f$. We will discuss them in the next sections.

3.3.2 Parsing of PL formulas

Here we describe how the parsing of a string, representing a PL formula. to an instance of `PLFormula`, works. The implementation of the PL formula parsing is provided by the class `PLParser`, declared in the module flloat/parser/pl.py. In Listing 3.6 is shown an example of use of this feature. The formula to be parsed is $A \wedge (\neg B \vee C) \wedge D$. The associated string that `PLParser` is able to understand is `A & (!B | C) & D`.

Listing 3.6. How to parse a PL formula: PLParser

```
from flloat.parser.pl import PLParser
parser = PLParser()
phi = parser("A␣&␣(!B␣|␣C)␣&␣D")
```

In Listing 3.7 are shown the implementation of PLParser and the associated lexer, PLLexer. They extends a more general definition of Parser and Lexer, implemented in the module flloat/base/parsing.py.

Listing 3.7. PLLexer and PLParser

```
class PLLexer(Lexer):

  def __init__(self):
    super().__init__()

  reserved = {
      Symbols.TRUE.value : 'TRUE',
      Symbols.FALSE.value : 'FALSE',
  }

  # List of token names. This is always required
  tokens = (
  'ATOM',
  'NOT',
  'AND',
  'OR',
  'IMPLIES',
  'EQUIVALENCE',
  'LPAREN',
  'RPAREN'
  ) + tuple(reserved.values())

  # Regular expression rules for simple tokens
  t_NOT         = sym2regexp(Symbols.NOT)
  t_AND         = sym2regexp(Symbols.AND)
  t_OR          = sym2regexp(Symbols.OR)
  t_IMPLIES     = sym2regexp(Symbols.IMPLIES)
  t_EQUIVALENCE = sym2regexp(Symbols.EQUIVALENCE)
  t_LPAREN      = sym2regexp(Symbols.ROUND_BRACKET_LEFT)
  t_RPAREN      = sym2regexp(Symbols.ROUND_BRACKET_RIGHT)



  def t_ATOM(self, t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    # Check for reserved words
    t.type = PLLexer.reserved.get(t.value, 'ATOM')
    return t

```

```

# Yacc
class PLParser(Parser):

  def __init__(self):
    lexer = PLLexer()
    precedence = (
      ('left', 'EQUIVALENCE'),
      ('right', 'IMPLIES'),
      ('left', 'OR'),
      ('left', 'AND'),
      ('right', 'NOT'),
      )
    super().__init__("pl", lexer.tokens, lexer, precedence)

  def p_formula_atom(self, p):
  """formula : ATOM
             | TRUE
             | FALSE"""
    if p[1]==Symbols.TRUE.value:
     p[0] = PLTrue()
    elif p[1]==Symbols.FALSE.value:
      p[0] = PLFalse()
    else:
      p[0] = PLAtomic(Symbol(p[1]))

  def p_formula_not(self, p):
  'formula : NOT formula'
    p[0] = PLNot(p[2])

  def p_formula_or(self, p):
  'formula : formula OR formula'
    p[0] = PLOr([p[1], p[3]])

  def p_formula_and(self, p):
  'formula : formula AND formula'
    p[0] = PLAnd([p[1], p[3]])

  def p_formula_implies(self, p):
  'formula : formula IMPLIES formula'
    p[0] = PLImplies([p[1], p[3]])

  def p_formula_equivalence(self, p):
  'formula : formula EQUIVALENCE formula'
    p[0] = PLEquivalence([p[1], p[3]])

  def p_formula_expression(self, p):
  'formula : LPAREN formula RPAREN'
```

```
    p[0] = p[2]
```

This feature depends from PLY, which provides utilities for both tokenization and parsing of a string. We do not go into details about how PLY works.

For the lexer part, we defied the available tokens for the parsing phase, as well as reserved words/symbols/regular expression for each of them (e.g. look at the regex for the `ATOM` token at line 34). The symbols used for the operators are defined in the module flloat/base/Symbols.py.

For the parsing part, we defined how to construct a syntax tree from the list of tokens. For each type of formula, we defined inductively a set of rules that, when activated, create the proper `PLFormula` object.

For example, consider line 55, where has been defined the rule to generate the atomic formulas. Depending on the type of token that activated the rule, we generate a `PLTrue`, `PLFalse` or a `PLAtomic` formula with the symbol provided by the tokenizer. Analogous considerations are made for other types of rules and formulas. Eventually, when there is no rule to apply, the entire syntax tree has been built. The object that we built "manually" (see Listing 3.1) now is built automatically by the parser.

This pattern is the same used for $\text{LTL}_f/\text{LDL}_f$ parsers.

3.3.3 PL formulas evaluation

In this section, we see how to compute the truth value of a `PLFormula`, given a propositional interpretation. We represent the propositional interpretation with the class `PLInterpretation`, that is trivially a set of `Symbols` (the ones that we consider true).

Listing 3.8. An example for PL formula truth evaluation

```python
from flloat.parser.pl import PLParser
parser = PLParser()
phi = parser("A␣&␣(!B␣|␣C)␣&␣D")

from flloat.semantics.pl import PLInterpretation

A_true = PLInterpretation.fromStrings(["A"])
print(phi.truth(A_true)) #prints False
ACD_true = PLInterpretation.fromStrings(["A", "C", "D"])
print(phi.truth(ACD_true)) #prints True
```

You can notice that we can easily define a `PLInterpretation` by using the static method `fromStrings` and providing a collection of strings corresponding to the propositions that we want to make *true*. The evaluation is made by the method `truth`, that is implemented by every `PLFormula`. As stated in Section 3.3.1, the actual implementation of the (trivial) procedures to evaluate the truths of logic operators are defined in flloat/base/truths.py, that are inherited by the different `PLFormulas`. The truth evaluation of atomic formulas such as `PLAtomic`, `PLTrue` and `PLFalse` are, respectively, in lines 28, 41 and 52 of Listing 3.5.

In Figure 3.3 is depicted the schema representing the relations between the classes used for logic formulas discussed until now. It is worth to notice the separation between the implementation of the syntax and the one of the semantics, from where the propositional formula classes inherit their functionalities. This organization is replicated in both LTL_f and LDL_f implementations.

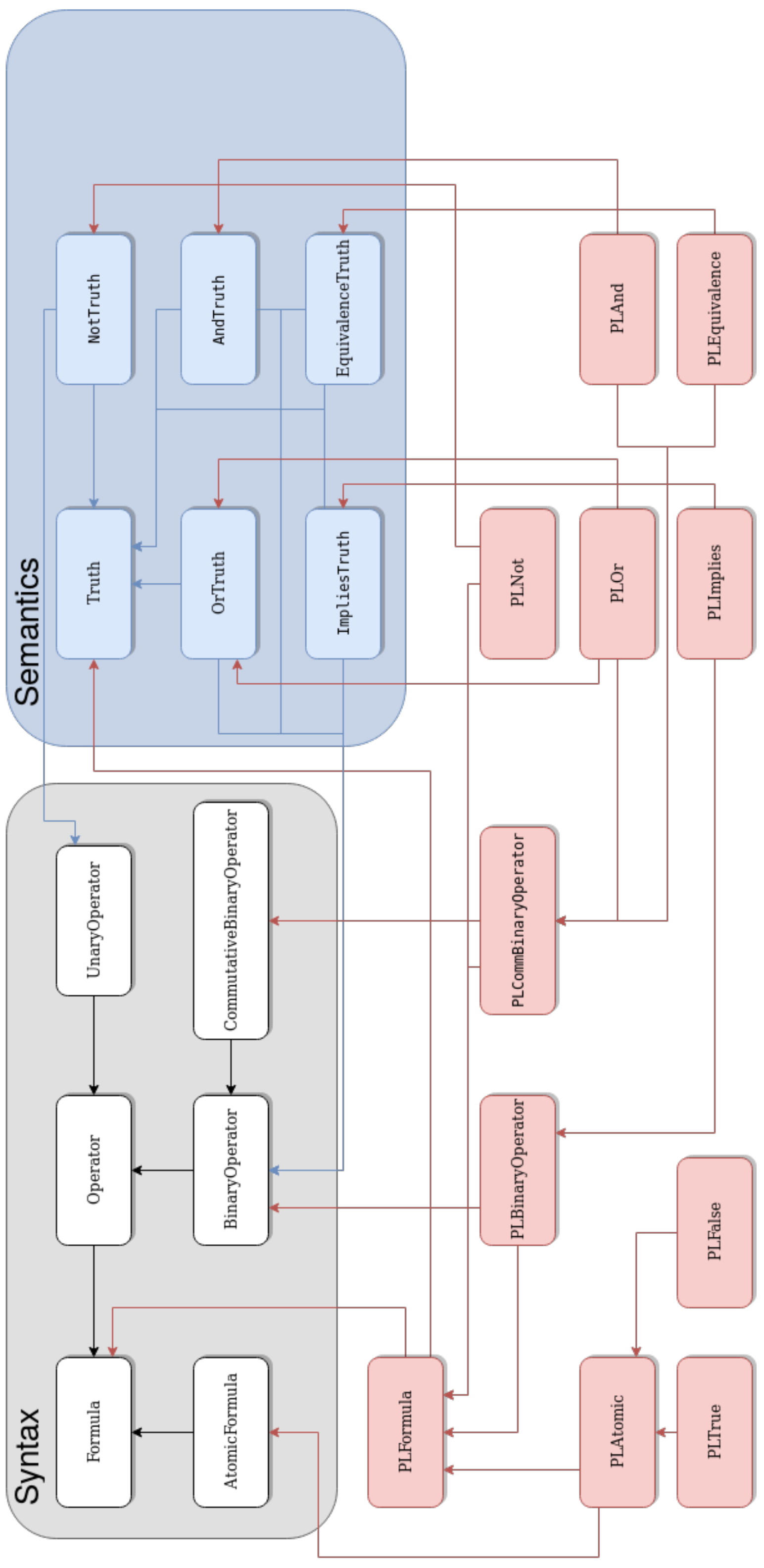


Figure 3.3. A schema representing the relations between the FLLOAT classes, that extends the one in Figure 3.2. The arrows mean "extends". In white the classes that implement how the syntax tree is built. In light blue the classes that implement the truth evaluation of operators. In red the classes that represent PL formulas. PL classes inherits from both syntax classes and semantics classes without implementing again both syntax and semantics functionalities.

3.4 LTL$_f$ formulas

As stated before, the software architecture to support LTL$_f$/LDL$_f$ formulas is pretty the same of the one presented in Section 3.3. The core syntax functionalities (i.e. how to build the syntax tree of a LTL$_f$/LDL$_f$ formula) are inherited from the classes defined in `flloat/base/Formula.py`, as described in Section 3.3.1. The semantics functionalities for propositional logic operator (e.g. And, Or, Not) are inherited from `Truths` classes declared in flloat/base/truths.py (see Section 3.3.3). Other ad-hoc data structures and classes are defined in order to support LTL$_f$/LDL$_f$ formulas.

In this section, we describe the LTL$_f$ part analogously to what we did in Section 3.3 for PL. In the next section, we do the same for LDL$_f$.

3.4.1 Specify LTL$_f$ formulas

In this section, we will see how to build LTL$_f$ formulas.

In Listing 3.9 is shown the construction of the main LTL$_f$ formulas.

Listing 3.9. Examples of LTL$_f$ formulas

```python
from flloat.base.Symbol import Symbol
from flloat.syntax.ltlf import LTLfAtomic, LTLfAnd,\
LTLfOr, LTLfNot, LTLfImplies, LTLfEquivalence,\
LTLfNext, LTLfWeakNext, LTLfUntil, LTLfRelease, \
LTLfAlways, LTLfEventually
A_sym, B_sym = Symbol("A"), Symbol("B")
A = LTLfAtomic(A_sym)
B = LTLfAtomic(B_sym)

# classic logic operators
A_and_B = LTLfAnd([A, B])
A_or_B = LTLfOr([A, B])
not_A = LTLfNot(A)
A_implies_B = LTLfImplies([A, B])
A_equivalence_B = LTLfEquivalence([A, B])

#temporal operators
next_A = LTLfNext(A)
wnext_A = LTLfWeakNext(A)
A_until_B = LTLfUntil([A, B])
A_release_B = LTLfRelease([A, B])
always_A = LTLfAlways(A)
eventually_A = LTLfEventually(A)
```

All the classes for construct LTL$_f$ formulas are declared in flloat/syntax/ltlf.py. The formula `LTLfAtomic` and the operators `LTLfAnd`, `LTLfOr`, `LTLfNot`, `LTLfImplies` and `LTLfEquivalence` are conceptually the same of the PL formulas described in Section 3.3.1. They extends `LTLfFormula` instead of `PLFormula` and inherits the syntax functionalities from the base classes in `flloat/base/Formula.py`.

The novelty is about the temporal formulas `LTLfNext`, `LTLfWeakNext`, `LTLfUntil`, `LTLfRelease`, `LTLfAlways` and `LTLfEventually`, each of them extending `LTLfTemporalFormula`, which in turn extends `LTLfFormula`. They inherit the syntax functionalities from `UnaryOperator` and `BinaryOperator`, in a straightforward way.

Observe that the operators `LTLfWeakNext`, `LTLfRelease`, `LTLfAlways` and `LTLfEventually` are *abbreviations*, in the sense that they can be rewritten by only using combinations of `LTLfNot`, `LTLfNext` and `LTLfUntil`. In order to reuse the code as much as possible, the relevant methods of these operators (such as `truth` or `to_nnf`, which put the formula in NNF) actually first *convert* the formula to an equivalent one by using only $\neg, \bigcirc, \mathcal{U}$, and then call the method on the newly created formula.

In order to do that, these classes extends the `BaseConvertibleFormula` declared in flloat/base/convertible.py (Listing 3.10). This concept is used also in the implementation of support for LDL$_f$ formulas.

Listing 3.10. The `BaseConvertibleFormula` abstraction, in flloat/base/convertible.py.

```
...

class ConvertibleFormula(Formula):
@abstractmethod
def _convert(self):
raise NotImplementedError

class BaseConvertibleFormula(ConvertibleFormula, Truth, NNF):
def truth(self, *args):
return self._convert().truth(*args)

def _to_nnf(self):
return self._convert().to_nnf()

def negate(self):
return self._convert().negate()

...
```

Indeed, by continuing from Listing 3.9:

```
print(wnext_A._convert())   #prints '!(X(!(A)))'
print(A_release_B._convert()) #prints '!((!(A) U !(B)))'
print(eventually_A._convert()) #prints '(true U A)'
print(always_A._convert())  #prints '!((true U !(A)))'
```

`LTLfFormula`: the base abstraction for LTL$_f$ formulas

In Listing 3.11 is shown the implementation of `LTLfFormula`, which is extended by every LTL$_f$ formula class. It extends the following classes (skipping the class `Formula`):

- `LTLfTruth` (line 1) which defines the method `truth` that takes in input a `FiniteTrace` (see Section 2.2.2 and Section 3.4.3) and a position from which evaluate the trace (by default 0, i.e. from the beginning).
- `NNF`, a class which defines the abstract method `to_nnf` that returns the same formula but in Negation Normal Form (NNF);

- Delta, a class which defines the abstract method _delta, (line 23) called from delta (line 10), that implements the delta function described in Section 2.5.1.

Every subclass implements those methods properly, by taking into account the definitions provided in Section 2.2.2 and Section 2.5.1.

Listing 3.11. The LTLfFormula abstraction (flloat/syntax/ltlf.py)

```
 1 class LTLfTruth(Truth):
 2   @abstractmethod
 3   def truth(self, i: FiniteTrace, pos: int = 0):
 4     raise NotImplementedError
 5
 6
 7 class LTLfFormula(Formula, LTLfTruth, NNF, Delta):
 8
 9   @lru_cache(maxsize=MAX_CACHE_SIZE)
10   def delta(self, i: PLInterpretation, epsilon=False):
11     f = self.to_nnf()
12     d = f._delta(i, epsilon)
13     if epsilon:
14       # By definition, if epsilon=True, then
15       #the result must be either PLTrue or PLFalse
16       # Now, the output is a Propositional Formula
17       # with only PLTrue or PLFalse as atomics
18       # Hence, we just evaluate the formula with a dummy PLInterpretation
19       d = PLTrue() if d.truth(None) else PLFalse()
20     return d
21
22   @abstractmethod
23   def _delta(self, i: PLInterpretation, epsilon=False):
24   """apply delta function, assuming that 'self' is a
25   LTLf formula in Negative Normal Form"""
26     raise NotImplementedError
27
28   @abstractmethod
29   def to_LDLf(self):
30     raise NotImplementedError
31
32   def to_automaton(self, labels:Set[Symbol]=None, on_the_fly=False,
33    determinize=False, minimize=True):
34     if labels is None:
35       labels = self.find_labels()
36     if on_the_fly:
37       return DFAOTF(self)
38     elif determinize:
39       return to_automaton(self, labels, minimize)
40     else:
41       return to_automaton_(self, labels)
```

For the `delta` function it has been used a LRU caching mechanism, provided from the Python Standard Library `functools`. This is needed in order to speed-up the expensive and repeated computation of the function `delta`.

At line 29 is defined the abstract method `to_LDLf`, which transform the current LTL$_f$ formula instance into an LDL$_f$ equivalent formula (by leveraging the equivalences shown in Section 2.4.2). Every specific LTL$_f$ formula class implements the procedure to transform itself into an equivalent LDL$_f$ formula. In practice, the root formula of the LTL$_f$ syntax tree computes the result by transforming each child into a LDL$_f$ formula, in such a way that the entire syntax tree is properly rewritten.

At line 33 it is defined the method `to_automaton` which transform the LTL$_f$ formula into an equivalent automaton. We discuss it in later sections.

3.4.2 Parsing of LTL$_f$ formulas

In this part we show how to use the LTL$_f$ parsing feature, analogously to what we did in Section 3.3.2.

In Listing 3.12 is shown an equivalent approach to build LTL$_f$ formulas, as we did in Listing 3.9.

Listing 3.12. Parsing of LTL$_f$ formulas

```
1  from flloat.parser.ltlf import LTLfParser
2
3  parser = LTLfParser()
4
5  # classic logic operators
6  A_and_B = parser("A␣&␣B")
7  A_or_B = parser("A␣|␣B")
8  not_A = parser("!A")
9  A_implies_B = parser("A␣->␣B")
10 A_equivalence_B = parser("A␣<->␣B")
11
12 #temporal operators
13 next_A = parser("X␣A")
14 wnext_A = parser("WX␣A")
15 A_until_B = parser("A␣U␣B")
16 A_release_B = parser("A␣R␣B")
17 always_A = parser("G␣A")
18 eventually_A = parser("F␣A")
```

The syntax of classic logic operators is the same of the `PLParser` presented in Section 3.3.2. The implementation of `LTLfParser` can be found in flloat/parser/ltlf.py. It has been implemented in the same fashion of `PLParser`, but with more reserved symbols (`X`, `WX`, `U`, `R`, `G`, and `F` for represent, repsectively, the LTL$_f$ operators $\circ, \bullet, \mathcal{U}, \mathcal{R}, \Box, \Diamond$) and their associated parsing rules. Combinations and arbitrary nesting of those operators with `"("` and `")"` are allowed.

Notice how rules defined from line 6 reflects the ones defined in Section 2.2.1.

Listing 3.13. An extract from flloat/parser/ltlf.py.

```
1  class LTLfParser(Parser):
```

```

...

 def p_formula(self, p):
     """formula : formula EQUIVALENCE formula
                  | formula IMPLIES formula
                  | formula OR formula
                  | formula AND formula
                  | formula UNTIL formula
                  | formula RELEASE formula
                  | EVENTUALLY formula
                  | ALWAYS formula
                  | NEXT formula
                  | WEAK_NEXT formula
                  | NOT formula
                  | TRUE
                  | FALSE
                  | ATOM"""
     if len(p) == 2:
       if p[1] == Symbols.TRUE.value:
         p[0] = LTLfTrue()
       elif p[1] == Symbols.FALSE.value:
         p[0] = LTLfFalse()
       else:
         p[0] = LTLfAtomic(Symbol(p[1]))
     elif len(p) == 3:
       if p[1] == Symbols.NEXT.value:
         p[0] = LTLfNext(p[2])
       elif p[1] == Symbols.WEAK_NEXT.value:
         p[0] = LTLfWeakNext(p[2])
       elif p[1] == Symbols.EVENTUALLY.value:
         p[0] = LTLfEventually(p[2])
       elif p[1] == Symbols.ALWAYS.value:
         p[0] = LTLfAlways(p[2])
       elif p[1] == Symbols.NOT.value:
         p[0] = LTLfNot(p[2])
     elif len(p) == 4:
       l, o, r = p[1:]
       if o == Symbols.EQUIVALENCE.value:
         p[0] = LTLfEquivalence([l, r])
       elif o == Symbols.IMPLIES.value:
         p[0] = LTLfImplies([l, r])
       elif o == Symbols.OR.value:
         p[0] = LTLfOr([l, r])
       elif o == Symbols.AND.value:
         p[0] = LTLfAnd([l, r])
       elif o == Symbols.UNTIL.value:
         p[0] = LTLfUntil([l, r])
```

```
      elif o == Symbols.RELEASE.value:
        p[0] = LTLfRelease([l, r])
      else:
        raise ValueError
    else:
      raise ValueError

  def p_expr_paren(self, p):
    """formula : LPAREN formula RPAREN"""
    p[0] = p[2]
```

3.4.3 LTL$_f$ formulas evaluation

The semantics of LTL$_f$ (as well as of LDL$_f$) is provided by finite traces π over a given set of propositional symbols $\mathcal{P}$, i.e. a sequence of propositional interpretations $\Pi \in 2^{\mathcal{P}}$.

FLLOAT allow to construct `FiniteTrace` instances in a easy way: providing to the static method `FiniteTrace.fromStringSets` a list of sets of strings, where each string represent a propositional symbol of interest. In Listing 3.14 an example about how to use `FiniteTrace` is given.

Listing 3.14. Defining a finite trace $\pi = \langle \{\}, \{A\}, \{A, B\} \rangle$ in FLLOAT.

```
from flloat.semantics.ldlf import FiniteTrace

t1 = FiniteTrace.fromStringSets([
  {},
  {"A"},
  {"A", "B"},
])
```

In Listing is shown how to evaluate LTL$_f$ formulas.

Listing 3.15. Some examples about how evaluate LTL$_f$ formulas.

```
from flloat.parser.ltlf import LTLfParser
from flloat.semantics.ldlf import FiniteTrace

parser = LTLfParser()

#temporal operators
next_A = parser("X␣A")
wnext_A = parser("WX␣A")
A_until_B = parser("A␣U␣B")
A_release_B = parser("A␣R␣B")
always_A = parser("G␣A")
eventually_A = parser("F␣A")

t1 = FiniteTrace.fromStringSets([
  {},
  {"A"},
  {"B"},
  {"A", "B"},
])

next_A.truth(t1) #True
next_A.truth(t1, 3) #False, at the end of the trace

wnext_A.truth(t1, 1) #False
wnext_A.truth(t1, 3) #True, at the end of the trace

A_until_B.truth(t1) #False
A_until_B.truth(t1, 3) #True

A_release_B.truth(t1) #False
A_release_B.truth(t1, 2) #True

always_A.truth(t1) #False
always_A.truth(t1, 3) #True

eventually_A.truth(t1) #True
eventually_A.truth(t1, 4) #False, beyond the end of the trace
```

For the LTL$_f$ classic logical operators, the implementation of the semantics is exactly the same of the one used for PL formulas (see Section 3.3.3).

For the LTL$_f$ temporal operators, the implementation is provided case by case following the definition that is given in Section 2.2.2.

3.5 LDL$_f$ formulas

In this section, we show how FLLOAT supports LDL$_f$ formulas, as we did Section 3.3 and Section 3.4.

3.5.1 Specify LDL$_f$ formulas

In this part we will see how to build LDL$_f$ formulas.

In Listing 3.16 is shown the construction of the main LDL$_f$ formulas.

Listing 3.16. Examples of LDL$_f$ formulas

```
from flloat.base.Symbol import Symbol
from flloat.syntax.ldlf import LDLfAtomic, LDLfAnd,\
LDLfOr, LDLfNot, LDLfImplies, LDLfEquivalence,\
LDLfBox, LDLfDiamond,\
LDLfLogicalTrue, LDLfLogicalFalse,\
RegExpPropositional, RegExpTest,\
RegExpUnion, RegExpSequence, RegExpStar
from flloat.parser.pl import PLParser

#propositional formulas
```

```
11 plparser = PLParser()
12 A = plparser("A")
13 B = plparser("B")
14 true = plparser("true") #PLTrue
15 false = plparser("false") #PLFalse
16 A_and_B = plparser("A␣&␣B")
17 A_or_B = plparser("A␣|␣B")
18 not_A = plparser("!A")
19
20 #regular expressions
21 regA = RegExpPropositional(A)
22 regB = RegExpPropositional(B)
23 regTrue = RegExpPropositional(true)
24 regAandB = RegExpPropositional(A_and_B)
25 seq_AB = RegExpSequence([
26 RegExpPropositional(A), RegExpPropositional(B)
27 ])
28 union_AB = RegExpUnion([
29 RegExpPropositional(A), RegExpPropositional(B)
30 ])
31 true_star = RegExpStar(regA)
32
33 #LDLf formulas
34 tt = LDLfLogicalTrue()
35 ff = LDLfLogicalFalse()
36 diamond_A_tt = LDLfDiamond(regA, tt)
37 diamond_B_tt = LDLfDiamond(regB, tt)
38 diamond_seqAB_tt = LDLfDiamond(seq_AB, tt)
39 diamond_unionAB_tt = LDLfDiamond(union_AB, tt)
40 diamond_testA_B = LDLfDiamond(RegExpTest(diamond_A_tt), diamond_B_tt)
41 eventually_A = LDLfDiamond(true_star, diamond_A_tt)
42 always_A = LDLfBox(true_star, diamond_A_tt)
43 last = LDLfBox(regTrue, ff)
```

The classes implementing the regular expressions are named `RegExp*` (at line 20), whereas the ones for LDL$_f$ formulas are named `LDLf*` (at line 33). They are all declared in flloat/syntax/ldlf.py. The inheritance schema is pretty similar to the one of LTL$_f$ formulas, with the difference that for the `LDLfBox` $[\varrho]\varphi$ and `LDLfDiamond` $\langle\varrho\rangle\varphi$ formulas we need a regular expression and a LDL$_f$ subformula, hence supporting the double-inductive nature of LDL$_f$.

In Listing 3.17 is shown the code for the main regular expressions abstraction `RegExpFormula` (at line 12) and the abstraction for LDL$_f$ temporal formulas `LDLfTemporalFormula` (at line 15). At line 35 and 23 there are the implementations of, respectively, the `LDLfBox` and `LDLfDiamond` formulas.

Notice the constructor of `LDLfTemporalFormula`, which takes a regular expression and a LDL$_f$ formula. Notice also the definition of the abstract methods for implement the delta function (Section 2.5.2). `deltaDiamond` (line 31) compute the delta function for diamond LDL$_f$ formulas, whereas `deltaBox` (line 44) compute the delta function for box LDL$_f$ formulas.

Listing 3.17. Regular Expression abstract class `RegExpFormula` and LDL$_f$ formulas `LDLfFormula`.

```python
class DeltaRegExp(ABC):
  @abstractmethod
  def deltaDiamond(self, f:LDLfFormula, i: PLInterpretation,
      epsilon=False):
    raise NotImplementedError

  @abstractmethod
  def deltaBox(self, f:LDLfFormula, i: PLInterpretation,
      epsilon=False):
    raise NotImplementedError

class RegExpFormula(Formula, RegExpTruth, NNF, DeltaRegExp):
  pass

class LDLfTemporalFormula(LDLfFormula):
  def __init__(self, r:RegExpFormula, f:LDLfFormula):
    super().__init__()
    self.r = r
    self.f = f

  ...

class LDLfDiamond(LDLfTemporalFormulaNNF, FiniteTraceTruth):
  temporal_brackets = "<>"

  def truth(self, i: FiniteTrace, pos: int):
    return any(self.r.truth(i, pos, j) and\
      self.f.truth(i, j) for j in range(pos, i.last()+1))
    # last + 1 in order to include the last step

  def _delta(self, i:PLInterpretation, epsilon=False):
    return self.r.deltaDiamond(self.f, i, epsilon)


class LDLfBox(ConvertibleFormula, LDLfTemporalFormulaNNF):
  temporal_brackets = "[]"

  def _convert(self):
    return LDLfNot(LDLfDiamond(self.r, LDLfNot(self.f)))

  def truth(self, i: FiniteTrace, pos: int):
    return self._convert().truth(i, pos)

  def _delta(self, i:PLInterpretation, epsilon=False):
    return self.r.deltaBox(self.f, i, epsilon)
```

3.5.2 Parsing of LDL$_f$ formulas

The same LDL$_f$ formulas shown in Listing 3.16 are equivalently expressed by using the `LDLfParser` declared in flloat/parser/ldlf.py, as shown in Listing 3.18.

Listing 3.18. Examples of LDL$_f$ parsing

```python
from flloat.parser.ldlf import LDLfParser

parser = LDLfParser()

#LDLf formulas
tt = parser("tt")
ff = parser("ff")
diamond_A_tt = parser("<A>tt")
diamond_B_tt = parser("<B>tt")
diamond_seqAB_tt = parser("<A;B>tt")
diamond_unionAB_tt = parser("<A+B>tt")
diamond_testA = parser("<(<A>tt?)>tt")
eventually_A = parser("<true*><A>tt")
always_A = parser("[true*]<A>tt")
end = parser("[true]ff")
```

Notice that the regular expression operators ; +,* , ? are denoted by the associated ASCII characters. The diamond operator brackets are denoted by `<>`, whereas the box operator brackets by `[]`.

In Listing 3.19 are reported the rules for the processing of LDL$_f$ tokens and the parsing of a `LDLfFormula`.

Listing 3.19. Parsing rules for LDL$_f$ (excerpts from flloat/parser/ldlf.py)

```python
...
class LDLfParser(Parser):

  def p_temp_formula(self, p):
  """temp_formula : temp_formula EQUIVALENCE temp_formula
                  | temp_formula IMPLIES temp_formula
                  | temp_formula OR temp_formula
                  | temp_formula AND temp_formula
                  | BOXLSEPARATOR path BOXRSEPARATOR temp_formula
                  | DIAMONDLSEPARATOR path DIAMONDRSEPARATOR temp_formula
                  | NOT temp_formula
                  | TT
                  | FF
                  | END
                  | LAST"""
    ...

  def p_path(self, p):
  """path : path UNION path
          | path SEQ path
          | path STAR
          | temp_formula TEST
          | propositional"""
    ...


  def p_propositional(self, p):
  """propositional : propositional EQUIVALENCE propositional
                   | propositional IMPLIES propositional
                   | propositional OR propositional
                   | propositional AND propositional
                   | NOT propositional
                   | FALSE
                   | TRUE
                   | ATOM"""
    ...

  def p_expr_paren(self, p):
  """temp_formula : LPAREN temp_formula RPAREN
    path          : LPAREN path RPAREN
    propositional : LPAREN propositional RPAREN"""
    p[0] = p[2]
```

Observe that `p_temp_formula` defines the parsing rules for LDL$_f$ formulas; `p_path` defines the parsing rules for regular expressions; `p_propositional` defines the rules for propositional formulas. `p_expr_paren` allow for arbitrary nesting of round brackets.

3.5.3 From LTL$_f$ to LDL$_f$

Notice that you can build an LDL$_f$ by first constructing an LTL$_f$ formula, as described in Section 3.4.1 and Section 3.4.2, and then transform it into an equivalent LDL$_f$ formula by calling the method `to_LDLf()`:

Listing 3.20. Specify LDL$_f$ formulas from LTL$_f$ formulas.

```
from flloat.parser.ltlf import LTLfParser
parser = LTLfParser()

next_A = parser("X(A)")
ldlf_next_A = next_A.to_LDLf()
print(ldlf_next_A.to_nnf())
#<true>((<A>(tt) & <true>(tt)))

wnext_A = parser("WX(A)")
ldlf_wnext_A = wnext_A.to_LDLf()
print(ldlf_wnext_A.to_nnf())
#[true]((<A>(tt) | [true](ff)))

always_A = parser("G(A)")
ldlf_always_A = always_A.to_LDLf()
```

```
print(ldlf_always_A.to_nnf())
#[(((<true>(tt))? ; true))*]((<A>(tt) | [true](ff)))
```

You can verify the correctness of the translation by looking at Section 2.4.1.

3.5.4 LDL$_f$ formulas evaluation

The truth evaluation of LDL$_f$ formula is similar to the one of LTL$_f$ (see Section 3.4.3). In Listing 4.2 we report a piece of code similar to Listing 3.15 but adapted for LDL$_f$ formulas.

Listing 3.21. Examples of LDL$_f$ truth evaluations

```
from flloat.parser.ldlf import LDLfParser
from flloat.semantics.ldlf import FiniteTrace

parser = LDLfParser()

#LDLf formulas
tt = parser("tt")
ff = parser("ff")
diamond_A_tt = parser("<A>tt")
diamond_B_tt = parser("<B>tt")
diamond_seqAB_tt = parser("<A;B>tt")
diamond_unionAB_tt = parser("<A+B>tt")
diamond_testA = parser("<(<A>tt?)>tt")
eventually_A = parser("<true*><A>tt")
always_A = parser("[true*]<A>tt")
end = parser("[true]ff")

t1 = FiniteTrace.fromStringSets([
  {},
  {"A"},
  {"B"},
  {"A", "B"},
])

tt.truth(t1) #True
ff.truth(t1) #False

diamond_A_tt.truth(t1) #False
diamond_A_tt.truth(t1, 1) #True

diamond_seqAB_tt.truth(t1) #False
diamond_seqAB_tt.truth(t1, 1) #False

diamond_unionAB_tt.truth(t1) #False
diamond_unionAB_tt.truth(t1, 1) #True

diamond_testA.truth(t1) #False
```

```python
diamond_testA.truth(t1, 1) #True

eventually_A.truth(t1) #True
eventually_A.truth(t1, 4) #False

always_A.truth(t1) #False
always_A.truth(t1, 4) #True
```

The implementation of `truth` for each type of formula is defined according to the Definition 2.7.

3.6 LTL$_f$/LDL$_f$ formula to automaton

The core feature of FLLOAT is to support the conversion from LTL$_f$/LDL$_f$ formula to automaton. We've seen how to do it theoretically by translation into NFA (Section 2.5.3), on-the-fly evaluation (Section 2.6.1) and direct computation of the DFA (Section 2.6.2).

The method to be used is `to_automaton`, implemented by every `LTLfFormula` and `LDLfFormula`, shown at line 33 of Listing 3.11. By different values of the flag parameters, it is possible to switch into every mode. The implementations of the algorithms and the data structures are in the module flloat/flloat.py.

FLLOAT uses the Python package Pythomata for automata support. However, we just explore the basic use cases with the Pythomata APIs.

In this section, we explain how to use FLLOAT in the aforementioned cases.

3.6.1 From LTL$_f$/LDL$_f$ formula to NFA

In Listing 3.22 is shown how to use the APIs to generate a NFA from a LTL$_f$/LDL$_f$ formula. For simplicity we use `LTLfFormula` objects, but the same examples hold also for `LDLfFormula` instances.

Listing 3.22. From LTL$_f$/LDL$_f$ to NFA

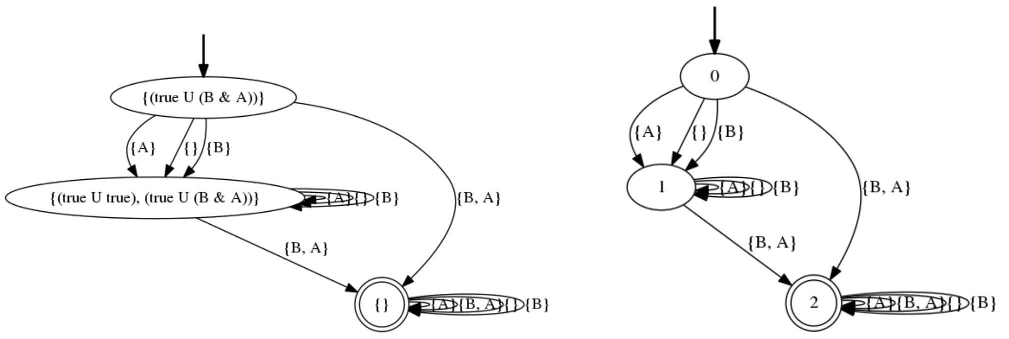
```python
1 from flloat.parser.ltlf import LTLfParser
2 parser = LTLfParser()
3 formula = parser("F(A␣&␣B)")
4
5 nfa = formula.to_automaton(determinize=False)
```

After the execution of these lines of code, in `nfa` is stored a `pythomata.base.NFA.NFA` object that represents an equivalent NFA. The method `LTLfFormula.to_automaton` at line 5 dispatches the execution to the function `to_automaton_` in flloat/flloat.py, which implements LDL$_f$2NFA (Algorithm 2.1).

Now you can print the NFA:

```python
nfa.to_dot("my_nfa")
nfa.map_to_int().to_dot("my_nice_nfa")
```

That generates SVG images similar to the one shown in Figure 3.4a and Figure 3.4b. They are equivalent, but the latter is "nicer" because does not print the meaning of the states, which might not be of interest. Notice how this representation is structurally identical to the one shown in 2.4.



(a) `nfa.to_dot("my_nfa")` (b) `nfa.map_to_int().to_dot("my_nice_nfa")`

Figure 3.4. Examples of conversion from LTL$_f$/LDL$_f$ to NFA and conversion into SVG images.

3.6.2 On the fly evaluation of LTL$_f$/LDL$_f$ formulas

FLLOAT supports also the on-the-fly evaluation of LTL$_f$/LDL$_f$ formulas, i.e. whether a finite trace satisfies an LTL$_f$/LDL$_f$ without building the entire automaton (see Section 2.6.1).

Listing 3.23. From LTL$_f$/LDL$_f$ to NFA

```
1 from flloat.parser.ltlf import LTLfParser
2 parser = LTLfParser()
3 formula = parser("F(A␣&␣B)")
4
5 dfaotf = formula.to_automaton(on_the_fly=True)
```

In Listing 3.23 it is shown how to use this feature. Notice that the method used, `to_automaton`, is the same used in Section 3.6.1 for the LDL$_f$2NFA algorithm, but with the flag `on_the_fly` set to `True`. In this case, 5 returns an object of the class flloat.flloat.DFAOTF, that implements the concept explained in Section 2.6.1.

Listing 3.24. The class `DFAOTF`, that implements on-the-fly LTL$_f$/LDL$_f$ formula evaluation.

```
class DFAOTF(Simulator):
"""DFA on the fly"""

  def __init__(self, f):
    self.f = f.to_nnf()
    self.reset()

  def reset(self):
    self.cur_state = frozenset([frozenset([self.f])])

  def word_acceptance(self, action_set_list:List[PLInterpretation]):
    self.reset()
    for a in action_set_list:
      self.make_transition(a)
```

```python
        return self.is_true()

    def is_true(self):
        return self._is_true(self.cur_state)

    @staticmethod
    def _is_true(Q):
        ...

    def get_current_state(self):
        return self.cur_state

    def make_transition(self, i:PLInterpretation):
        self.cur_state = self._make_transition(self.cur_state, i)

    @staticmethod
    def _make_transition(Q, i: PLInterpretation):
        ...
```

In Listing 3.24 it is shown the implementation of the class `DFAOTF`, that you can find in the module flloat/flloat.py. It extends `Simulator`, which is a Pythomata abstract class for inherit `word_acceptance` interface method. For the sake of brevity, we omitted some methods that have many lines. In the following we briefly describe the main methods:

- `reset()` set the current state to the initial macro state $\{\{\varphi\}\}$ notice that here (and in many other places) we used the data structure `frozenset`, which is an hashable set. This is useful for deal with *sets of sets* (e.g. macro states);
- `__init__` is the constructor that takes in input a $\text{LTL}_f/\text{LDL}_f$ formula, compute its NNF and reset the state to the initial state `reset`;
- `is_true()`, whose implementation depends on `_is_true()` returns `True` if the current state contains the empty conjunction $\emptyset$, which stand for *true* (see Section 2.6.1 for more details);
- `word_acceptance` takes in input a finite trace (i.e. a list of propositional interpretations) and returns `True` if the last state contains the empty conjunction (see `_is_true()`);
- `make_transition` and `_make_transition` implements the computation of the next macro state, given a macro state and a propositional interpretation. In other words, they implements the progression rule for macro states, stated in Equation 2.19.

Typical uses of a `DFAOTF` instance are evaluation of finite traces. By continuing with Listing 3.23, you can do it manually:

```python
from flloat.semantics.pl import PLInterpretation
from flloat.semantics.pl import PLFalseInterpretation

A = PLInterpretation.fromStrings(["A"])
B = PLInterpretation.fromStrings(["B"])
```

```
empty = PLFalseInterpretation()
AB = PLInterpretation.fromStrings(["A", "B"])

dfaotf.make_transition(empty)
dfaotf.make_transition(A)
dfaotf.make_transition(B)
dfaotf.is_true() #False
dfaotf.make_transition(AB)
dfaotf.is_true() #True
```

Or by simply calling `word_acceptance` over a list of `PLInterpretation`:

```
from flloat.semantics.ldlf import FiniteTrace

t1 = FiniteTrace.fromStringSets([
{},
{"A"},
{"B"},
{"A", "B"},
])

dfaotf.word_acceptance(t1.trace) #True
```

3.6.3 From LTL$_f$/LDL$_f$ formula to DFA

Here we describe how FLLOAT supports the transformation from LTL$_f$/LDL$_f$ formulas to DFA, i.e. the LDL$_f$2DFA algorithm (Algorithm 2.2).

We adapt Listing 3.22 by simply using the same method `to_automaton` but setting up the flag `determinze` to `True`.

Listing 3.25. From LTL$_f$/LDL$_f$ to DFA

```
1 from flloat.parser.ltlf import LTLfParser
2 parser = LTLfParser()
3 formula = parser("F(A␣&␣B)")
4
5 dfa = formula.to_automaton(determinize=True, minimize=False)
```

After the execution of these lines of code, in `dfa` is stored a `pythomata.base.DFA.DFA` object that represents an equivalent DFA. The method `LTLfFormula.to_automaton` at line 5 dispatches the execution to the function `to_automaton` in flloat/flloat.py, which implements LDL$_f$2DFA (Algorithm 2.1).

Now you can print the DFA:

```
dfa.to_dot("my_dfa")
```

Quite often is better to obtain the minimized automaton:

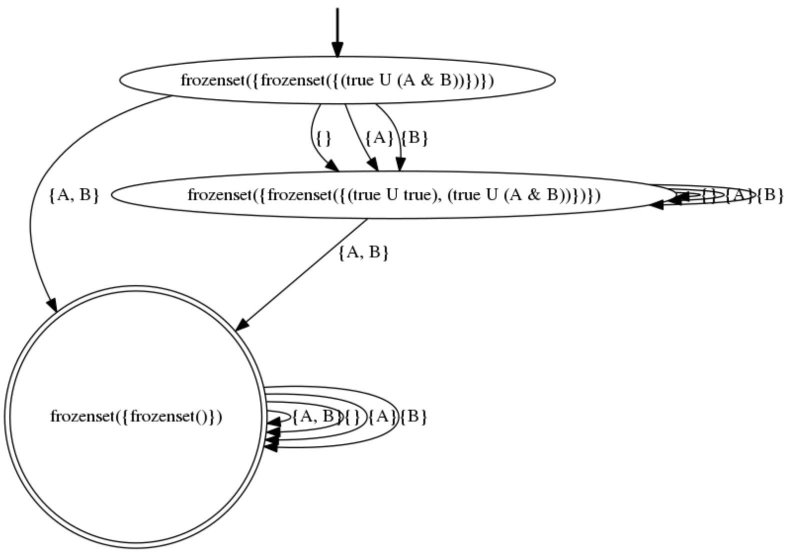
```
dfa_min = formula.to_automaton(determinize=True, minimize=True)
dfa.to_dot("my_nice_dfa")
```

By simply calling the same method but with the flag `minimize` set to `True`.

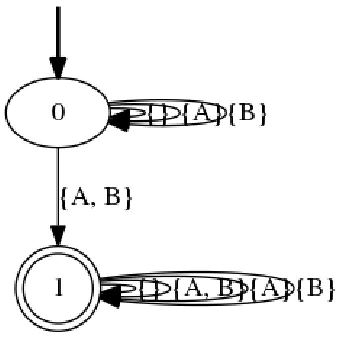
Another equivalent approach is:

```
nfa = formula.to_automaton(determinize=False)
dfa = nfa.determinize()
dfa_min = dfa.minimize().trim()
dfa.to_dot("my_nice_dfa")
```

In Figure 3.5a and Figure 3.5b are shown, respectively, the conversion to SVG images of the DFA not minimized and the one minimized. Notice how this representation is structurally identical to the one shown in 2.5.


frozenset({frozenset({(true U (A & B))})})
{}
{A}
{B}
{A, B}
frozenset({frozenset({(true U true), (true U (A & B))})})
{}{A}{B}
{A, B}
frozenset({frozenset()})
{A, B}{}{A}{B}

(a) `dfa.to_dot("my_dfa")`



(b) `dfa_min.to_dot("my_nice_dfa")`

Figure 3.5. Examples of conversion from LTL$_f$/LDL$_f$ to DFA and conversion into SVG images.

3.7 Conclusion

In this chapter, we presented the FLLOAT Python package and its main features. We've seen the package structure, how FLLOAT supports PL and LTL$_f$/LDL$_f$ formulas, and the conversion to automata from LTL$_f$/LDL$_f$ formulas.

Chapter 4

RL for LTL$_f$/LDL$_f$ Rewards

In this chapter, we discuss Reinforcement Learning with non-Markovian rewards specified by LTL$_f$/LDL$_f$ formulas. The main contribution of this chapter is to devise a natural extension of the construction proposed in (Brafman et al., 2018) for a reinforcement learning task. That is, we show that it is possible to do reinforcement learning for non-Markovian rewards, expressed in LTL$_f$/LDL$_f$ formulas, by expanding the state space of the agent from the automata associated to the LTL$_f$/LDL$_f$ reward formulas.

In the first section we summarize the framework of Reinforcement Learning. Then we introduce the foundational topics of MDP models and the Temporal Difference Learning. We explain what is an NMRDP and explain the rationale of the main solutions in the literature. Then we explain the result in (Brafman et al., 2018) and extend this work to the Reinforcement Learning domain.

4.1 Reinforcement Learning

Reinforcement Learning (Sutton and Barto, 1998) is a sort of optimization problem where an *agent* interacts with an *environment* and obtains a *reward* for each action he chooses and the new observed state. The task is to maximize a numerical reward signal obtained after each action during the interaction with the environment. The agent does not know a priori how the environment works (i.e. the effects of his actions), but he can make observations in order to know the new state and the reward. Hence, learning is made in a *trial-and-error* fashion. Moreover, it is worth to notice that in many situation reward might not be affected only from the last action but from an indefinite number of the previous actions. In other words, the reward can be *delayed*, i.e. the agent should be able to foresee the effect of his actions in terms of future expected reward. Figure 4.1 represent the interaction between the agent and the environment in this setting.

In the next subsections, we introduce some of the classical mathematical frameworks for RL: Markov Decision Process (MDP) and Non-Markovian Reward Decision Process (NMRDP).

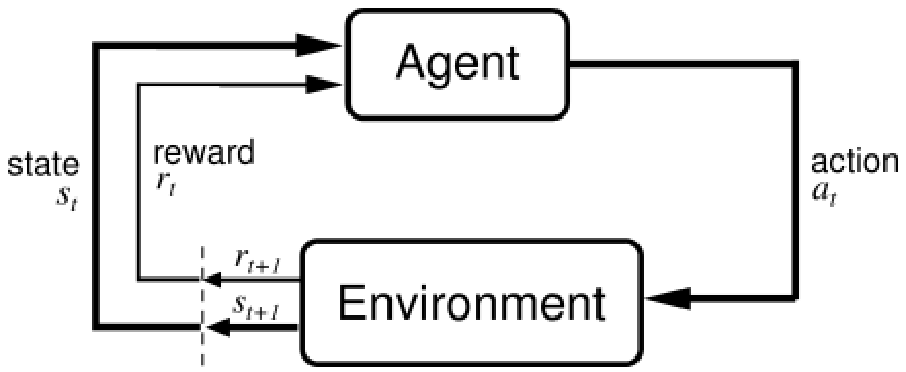


Figure 4.1. The agent and its interaction with the environment in Reinforcement Learning

4.2 Markov Decision Process (MDP)

A Markov Decision Process (MDP) $\mathcal{M}$ is a tuple $\langle S, A, T, R, \gamma \rangle$ containing a set of *states* S, a set of *actions* A, a *transition function* $T : S \times A \to Prob(S)$ that returns for every pair state-action a probability distribution over the states, a *reward function* $R : S \times A \times S \to \mathbb{R}$ that returns the reward received by the agent when he performs action a in s and transitions in s', and a *discount factor* γ, with $0 \leq \gamma \leq 1$, that indicates the present value of future rewards. With $T(s, a, s')$ we denote the probability to end in state s' given the action a from s.

The discount factor γ deserves some attention. Its value highly influences the MDP, its solution, and how the agent interprets rewards. Indeed, if $\gamma = 0$, we are in the pure *greedy* setting, i.e. the agent is shortsighted and looks only at the reward that it might obtain in the next step, by doing a single action. The higher γ, the longer the sight horizon, or the foresight, of the agent: the far rewards are taken into account for the current action choice. If $\gamma < 1$ we are in the *finite horizon* setting: namely, the agent is intrinsically motivated to obtain rewards as fast as possible, depending on how γ is far from 1. When $\gamma = 1$ we are in the *infinite horizon* setting, which means that the agent considers far rewards as they can be obtained in the next step. In other words, we may think the agent as *immortal*, since the time the agent spend to reach rewards does not matter anymore.

A *policy* $\rho : S \to A$ for an MDP $\mathcal{M}$ is a mapping from states to actions, and represents a solution for $\mathcal{M}$. Given a sequence of rewards $R_{t+1}, R_{t+2}, \ldots, R_T$, the *expected discounted return* G_t at time step t is defined as:

$$G_t \coloneqq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{4.1}$$

where can be $T = \infty$ and $\gamma = 1$ (but not both).

The *value function* of a state s, the *state-value function* $v_\rho(s)$ is defined as the expected return when starting in s and following policy ρ, i.e.:

$$v_\rho(s) \coloneqq \mathbb{E}_\rho[G_t | S_t = s], \forall s \in S \tag{4.2}$$

Similarly, we define q_ρ, the *action-value function for policy ρ*, as:

$$q_\rho(s,a) \coloneqq \mathbb{E}_\rho[G_t|S_t = s, A_t = a],, \forall s \in S, \forall a \in A \tag{4.3}$$

Notice that we can rewrite 4.2 and 4.3 recursively, yielding the *Bellman equation*:

$$v_\rho(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma v(s')] \tag{4.4}$$

where we used the definition of the transition function:

$$T(s,a,s') = P(s'|s,a) \tag{4.5}$$

We define the *optimal state-value function* and the *optimal action-value function* as follows:

$$v^*(s) \coloneqq \max_\rho v_\rho(s), \forall s \in S \tag{4.6}$$

$$q^*(s,a) \coloneqq \max_\rho q_\rho(s,a), \forall s \in S, \forall a \in A \tag{4.7}$$

Notice that with 4.6 and 4.7 we can show the correlation between $v^*_\rho(s)$ and $q^*_\rho(s,a)$:

$$q^*(s,a) = \mathbb{E}_\rho[R_{t+1} + \gamma v^*_\rho(S_{t+1})|S_t = s, A_t = a] \tag{4.8}$$

We can define a partial order over policies using value functions, i.e. $\forall s \in S. \rho \geq \rho' \iff v_\rho(s) \geq v_{\rho'}(s)$. Now we give the definition of optimal policy:

Definition 4.1. An *optimal policy* ρ^* is a policy such that $\rho^* \geq \rho$ for all ρ.

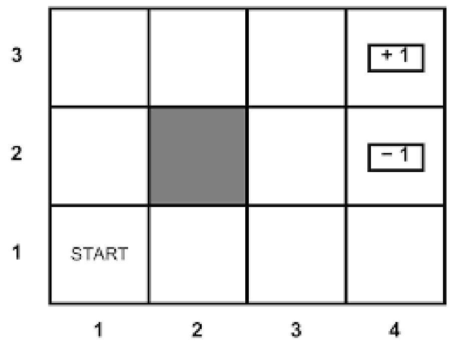
Given an MDP $\mathcal{M}$, a typical reinforcement learning problem is the following: find an optimal policy for $\mathcal{M}$, without knowing T and R. Notice that instead of explicit specification of the transition probabilities and rewards, the transition probabilities are accessed through a simulator that is restarted many times from a fixed or uniformly random initial state $s_0 \in S$. We call this way of structuring the learning process *episodic reinforcement learning*. Usually, in episodic reinforcement learning, we require the presence of one or more *goal states* where the simulation of the MDP ends and the task is considered completed, or a maximum time limit T for the number of actions that can be taken by the agent in one single episode, and the overcoming of T determines the end of the episode. Optionally, *failure states* can be also defined, where the episode ends similarly to goal states, but the task is considered failed.

Examples

Many dynamic systems can be modeled as Markov Decision Processes.

Example 4.1 (Gridworld). Perhaps the most simple MDP used as a toy example is *Gridworld*, depicted in Figure 4.2. There are 3×4 cells, i.e. states of the MDP $S = \{s_{11}, s_{12}, \dots, s_{34}\} \setminus \{s_{22}\}$. The agent can do four actions: $A = \{Right, Left, Up, Down\}$. The initial state is fixed and is $s_0 = s_{11}$ and the agent can move in any of the adjacent and free cells from the current state. Assuming an episodic task, the goal is to reach s_{34}, and s_{24} represent a failure state. The state transition function T can be *deterministic*, i.e. the agent always succeeds in performing actions, or non-deterministic, i.e. the effect of an action is determined by the probabilistic distribution returned by $T(s, a)$. An example of non-deterministic T is to give 90% of success (the agent moves in the chosen direction) and 10% of fail (the agent moves at either the right or left angle to the intended direction). If the move would make the agent walk into a wall (borders of the grid and s_{22}), the agent stays in the same place as before. The reward function $R(s, a, s')$ is defined as -1 if $s' = s_{24}$, as 1 if $s' = s_{34}$, and -0.01 otherwise. The small negative reward given at each transition is a popular mean for reward function design: it is called *step reward* and its purpose is to encourage the agent to finish the episode as fast as possible, with a priority proportional to the absolute value of the reward. The discount factor γ should be strictly higher than 0 because more than one step is needed to reach the goal state.

Figure 4.2. The Gridworld environment



An example of an optimal policy is shown in Figure 4.3. As the reader can notice, the arrows represent the action that should be taken in a certain cell, in order to maximize the expected return. We observe that the optimal action in s_{13}, according to the policy, is not the one to take the shortest path to the goal, i.e. the *Up* action . This is because there is a small probability to ens in s_{24}, the failure state, and be punished with a high negative reward. In terms of expected reward, it is better to take the longer path, at the price of collect small negative rewards, but avoiding the risk to fail miserably.

Example 4.2 (Breakout). Breakout is a well-known arcade video game developed by Atari. In this work, we implemented a clone of the original Breakout. Figure 4.4 shows a screenshot of the game. On the screen, there is a paddle at the bottom, many bricks at the top arranged in a grid layout with n rows and m columns (in the figure $3 \times 3 = 9$ bricks), and a ball that is free to move across the screen. The ball bounces when it hits a wall, a brick or the paddle. When the ball hits a brick, that brick is broke down and is removed from the screen. The paddle (the agent) can move left, move right or do nothing. The *goal* is to remove all the bricks while avoiding that the paddle misses the rebound of the ball (*failure*).

The relevant features are: position of the paddle p_x, position of the ball b_x, b_y, speed of the ball v_x, v_y and status of each brick (booleans) b_{ij}. This features of the system gives all the needed information to predict the next state from the current state. Hence we can build an MDP where: S is the set of all the possible values of the sequence of features $\langle p_x, b_x, b_y, v_x, v_y, b_{11}, \dots, b_{nm}\rangle$, $A = \{Right, Left, Noop\}$, transition function T determined by the rules of the game. We give reward $R(s, a, s') = 10$ if a particular

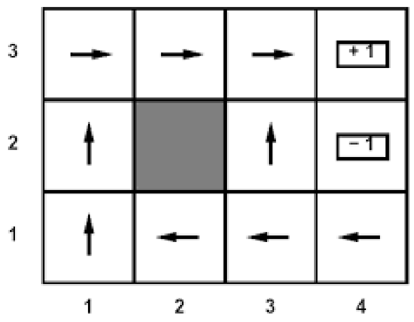


Figure 4.3. An example of optimal policy for the Gridworld environment

.

brick in s' has been removed for the first time, plus 100 if that brick was the last (i.e. goal reached).

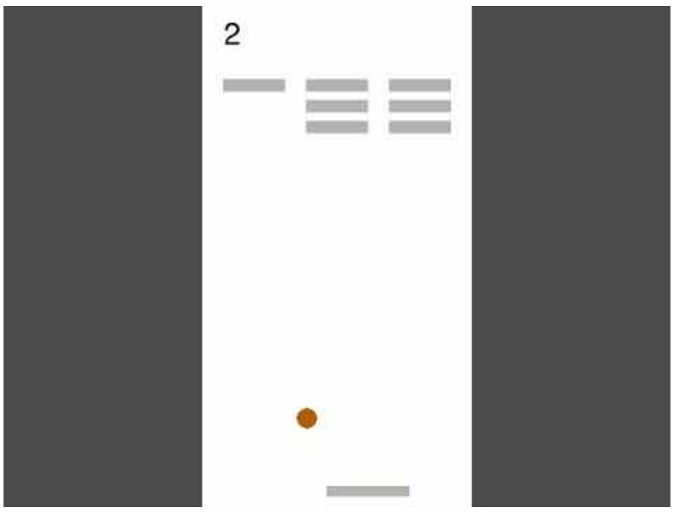


Figure 4.4. A screenshot from the video game BREAKOUT

.

Violation of the Markov property considering a smaller set of features: Notice that considering a strict subset of the set of features for S leads to violating the Markov property of T. Indeed, consider the case when we remove v_x and v_y from the set of features. In this setting, we removed the pieces of information about the dynamics of the system. More precisely, we cannot predict, knowing only the current state, the value of the features b_x and b_y for the next step, because we do not know where the ball is going (up-left, down-right and so on). In order to correctly predict the next position of the ball, we should know whether earlier in the episode the ball was coming from the bottom or from the top. But this fact clearly shows that the Markovian assumption is violated. Similar arguments apply in the case where we remove the status of the bricks $b_{11}, \ldots, b_{nm}$: indeed, if the ball in the next step is near to a brick, knowing about the status of the brick is determinant to predict if the ball will continue its trajectory (the case when the brick is absent) or it will break down the brick and bounce, changing the direction of its motion (the case when the brick is present).

4.3 Temporal Difference Learning

Temporal difference learning (TD) (Sutton, 1988) refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods sample from the environment, like Monte Carlo (MC) methods, and perform updates based on current estimates, like dynamic programming methods (DP) (Bellman, 1957). We do not discuss MC and DP methods here.

Q-Learning (Watkins, 1989; Watkins and Dayan, 1992) and Sarsa are such a methods. They update $Q(s,a)$, i.e. the estimation of $q^*(s,a)$ at each transition $(s,a) \to (s',r)$. The update rule is the following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta \tag{4.9}$$

where δ is the *temporal difference*. In Sarsa, it is defined as:

$$\delta = r + \gamma Q(s',a') - Q(s,a) \tag{4.10}$$

whereas in Q-Learning:

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a) \tag{4.11}$$

TD(λ) is an algorithm which uses *eligibility traces*. The parameter λ refers to the use of an eligibility trace. The algorithm generalizes MC methods and TD learning, obtained respectively by setting $\lambda = 1$ and $\lambda = 0$. Intermediate values of λ yield methods that are often better of the extreme methods. Q-Learning and Sarsa that has been shown before can be rephrased with this new formalism as Q-Learning(0) and Sarsa(0), special cases of Watkin's Q(λ) and Sarsa(λ) respectively. In this setting, Equation 4.9 is modified as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a) \tag{4.12}$$

Where $e(s,a) \in [0,1]$, the *eligibility of the pair* (s,a), determines how much the temporal difference δ should be weighted. Sarsa(λ) is reported in Algorithm 4.1, whereas Watkin's Q(λ) in Algorithm 4.2, both in the variants using *replacing eligibility traces* (see line 9 and line 10, respectively).

4.4 Non-Markovian Reward Decision Process (NMRDP)

For some goals, it might be the case that the Markovian assumption of the reward function R – that reward depends only on the current state, and not on history – does not hold. Indeed, for many problems, it is not effective that the reward is limited to depend only on a single transition (s,a,s'); instead, it might be extended to depend on *trajectories* (i.e. $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n\rangle$), e.g. when we want to reward the agent for some (temporally extended) behaviors, opposed to simply reaching certain states.

Algorithm 4.1. Sarsa(λ) (Singh and Sutton, 1996)

```
1: Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$ for all $s, a$
2: **repeat**{for each episode}
3:     initialize $s$
4:     Choose $a$ from $s$ using policy derived from $Q$ (e.g. $e$-greedy)
5:     **repeat**{for each step of episode}
6:         Take action $a$, observe reward $r$ and new state $s'$
7:         Choose $a'$ from $s'$ using policy derived from $Q$
8:         $\delta \leftarrow r + \gamma Q(s', a') - Q(s,a)$
9:         $e(s,a) \leftarrow 1$                                  ▷ replacing traces
10:        **for all** $s, a$ **do**
11:            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
12:            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
13:        **end for**
14:        $s \leftarrow s'$, $a \leftarrow a'$
15:    **until** state $s$ is terminal
16: **until**
```

Algorithm 4.2. Watkin's Q(λ) (Watkins, 1989)

```
1: Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$ for all $s, a$
2: **repeat**{for each episode}
3:     initialize $s$
4:     Choose $a$ from $s$ using policy derived from $Q$ (e.g. $e$-greedy)
5:     **repeat**{for each step of episode}
6:         Take action $a$, observe reward $r$ and new state $s'$
7:         Choose $a'$ from $s'$ using policy derived from $Q$ (e.g. $e$-greedy)
8:         $a^* \leftarrow \arg\max_a Q(s', a)$ (if $a'$ ties for max, then $a^* \leftarrow a'$)
9:         $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s,a)$
10:        $e(s,a) \leftarrow 1$                                  ▷ replacing traces
11:        **for all** $s, a$ **do**
12:            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
13:            **if** $a' = a^*$ **then**
14:                $e(s,a) \leftarrow \gamma\lambda e(s,a)$
15:            **else**
16:                $e(s,a) \leftarrow 0$
17:            **end if**
18:            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
19:        **end for**
20:        $s \leftarrow s'$, $a \leftarrow a'$
21:    **until** state $s$ is terminal
22: **until**
```

This idea of rewarding behaviours has been proposed by (Bacchus et al., 1996) where they defined a new mathematical model, namely Non-Markovian Reward Decision Process (NMRDP), and showed how to construct optimal policies in this case.

In the next subsections, we give the main definitions to reason in this new setting. Then we show the solution proposed in (Bacchus et al., 1996).

4.4.1 Preliminaries

Now follows the definition of NMRDP, which is similar to the MDP definition given in Section 4.2.

Definition 4.2. A Non-Markovian Reward Decision Process (NMRDP) (Bacchus et al., 1996) $\mathcal{N}$ is a tuple $\langle S, A, T, \bar{R}, \gamma\rangle$ where S, A, T and γ are defined as in the MDP, and $\bar{R} : S^* \to \mathbb{R}$ is the *non-Markovian reward function*, where $S^* = \{\langle s_0, s_1, \dots, s_n\rangle_{n\geq 0, s_i \in S}\}$ is the set of all the possible traces, i.e. projection of trajectories $\langle s_0, a_0, \dots, s_{n-1}, a_{n-1}, s_n\rangle$

Given a trace $\pi = \langle s_0, s_1, \dots, s_n\rangle$, the *value of* π is:

$$v(\pi) = \sum_{i=1}^{|\pi|} \gamma^{i-1} \bar{R}(\langle s_0, s_1, \dots, s_n\rangle) \tag{4.13}$$

where $|\pi|$ denotes the number of transitions (i.e. of actions).

The policy $\bar{\rho}$ in this setting is defined over sequences of states, i.e. $\bar{\rho} : S^* \to A$. The *value of* $\bar{\rho}$ given an initial state s_0 is defined as:

$$v^{\bar{\rho}}(s) = \mathbb{E}_{\pi\sim\mathcal{N},\bar{\rho},s_0}[v(\pi)] \tag{4.14}$$

i.e. the expected value in state s considering the distribution of traces defined by the transition function of $\mathcal{N}$, the policy $\bar{\rho}$ and the initial state s_0.

We are interested in two problems, that we will study in the next sections:

- Find an optimal (non-Markovian) policy $\bar{\rho}$ for an NMRDP $\mathcal{N}$ (Definition 4.2);
- Define the non-Markovian reward function for the domain of interest.

4.4.2 Find an optimal policy $\bar{\rho}$ for NMRDPs

The key difficulty with non-Markovian rewards is that standard optimization techniques, most based on Bellman's (Bellman, 1957) dynamic programming principle, cannot be used. Indeed, this requires one to resort to optimization over a policy space that maps histories (rather than states) into actions, a process that would incur a great computational expense. (Bacchus et al., 1996) give the definition of a decision problem *equivalent* to an NMRDP in which the rewards are Markovian. This construction is the key element to solve our problem, i.e. find an optimal policy for an NMRDP.

Equivalent MDP

Now we give the definition of *equivalent* MDP of an NMRDP, and state an important result.

Definition 4.3 (Bacchus et al. (1996))**.** An NMRDP $\mathcal{N} = \langle S, A, T, \bar{R}, \gamma\rangle$ is *equivalent* to an extended MDP $\mathcal{M} = \langle S', A, T', R', \gamma\rangle$ if there exist two functions $\tau : S' \to S$ and $\sigma : S \to S'$ such that

1. $\forall s \in S : \tau(\sigma(s)) = s$;
2. $\forall s_1, s_2 \in S$ and $s'_1 \in S'$: if $T(s_1, a, s_2) > 0$ and $\tau(s'_1) = s_1$, there exists a unique $s'_2 \in S'$ such that $\tau(s'_2) = s_2$ and $T'(s'_1, a, s'_2) = T(s_1, a, s_2)$;
3. For any feasible trace $\langle s_0, s_1, \ldots, s_n\rangle$ of $\mathcal{N}$ and $\langle s'_0, s'_1, \ldots, s'_n\rangle$ of $\mathcal{M}$ associated to the trajectories $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n\rangle$ and $\langle s'_0, a_0, \ldots, s'_{n-1}, a_{n-1}, s'_n\rangle$, such that $\tau(s'_i) = s_i$ and $\sigma(s_0) = s'_0$, we have $R(\langle s_0, s_1, \ldots, s_n\rangle) = R'(s_{n-1}, a_{n-1}, s'_n)$.

Given the Definition 4.3, we give the definition of corresponding policy:

Definition 4.4 (Bacchus et al. (1996))**.** Let $\mathcal{N}$ be an NMRDP and let $\mathcal{M}$ be the equivalent MDP as defined in Definition 4.3. Let ρ be a policy for $\mathcal{M}$. The *corresponding policy* for $\mathcal{N}$ is defined as $\bar{\rho}(\langle s_0, \ldots, s_n\rangle) = \rho(s'_n)$, where for the sequence $\langle s'_0, \ldots, s'_n\rangle$ we have $\tau(s'_i) = s_i$ $\forall i$ and $\sigma(s_0) = s'_0$

From definitions 4.3 and 4.4, and since that for all policy ρ of $\mathcal{M}$ the corresponding policy $\bar{\rho}$ of $\mathcal{N}$ is such that $\forall s. v_\rho(s) = v_{\bar{\rho}}(\sigma(s))$, the following theorem holds:

Theorem 4.1 (Bacchus et al. (1996))**.** *Let ρ be an optimal policy for MDP $\mathcal{M}$. Then the corresponding policy is optimal for NMRDP $\mathcal{N}$.*

The Theorem 4.1 allows us to learn an optimal policy $\bar{\rho}$ for NMRDP by learning a policy ρ over an equivalent MDP, which can be done by resorting on any off-the-shelf algorithm (e.g. see Section 4.3). Moreover, obtaining the corresponding policy for the original NMRDP is straightforward, although in practice is not needed, since it is enough to run the policy ρ over the MDP.

In other words, the problem of finding an optimal policy for an NMRDP reduces to find an optimal policy for an equivalent MDP such that Condition 1, 2 and 3 of Definition 4.3 hold.

4.4.3 Define the non-Markovian reward function $\bar{R}$

To reward agents for (temporally extended) behaviours, as opposed to simply reaching certain states, we need a way to specify rewards for specific trajectories through the state space. Specifying a non-Markovian reward function explicitly is quite hard and unintuitive, impossible if we are in an infinite-horizon setting. Instead, we can define *properties* over trajectories and reward only the ones which satisfy some of them, in contrast to enumerate all the possible trajectories.

Temporal logics presented in Section 2.1 gives an effective way to do this. Indeed, in order to speak about a desired behavior, i.e. fulfillment of properties that might change over time, we can define a *formula* φ (or more formulas) in some suited temporal logic formalism semantically defined over trajectories π, speaking about a set of properties $\mathcal{P}$ such that each state $s \in S$ is associated to a set of propositions ($S \subseteq 2^{\mathcal{P}}$). In this way, a trajectory $\pi = \langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n\rangle$ is rewarded with r_i iff $\pi \models \varphi_i$, where r_i is the reward value associated to the fulfillment of behaviours signified by φ_i.

4.4.4 Using PLTL

In (Bacchus et al., 1996) the temporal logic formalism is *Past Linear Temporal Logic* (PLTL), which is a past version of LTL (Section 2.1). As explained before, using the declarativeness of PLTL, is possible to specify the desired behaviour (expressed in terms of the properties $\mathcal{P}$) that should be satisfied by the experienced trajectories and reward only them, hence obtaining a non-Markovian reward function. More formally, given a finite set Φ of PLTL *reward formulas*, and for each $\phi_i \in \Phi$ a real-valued reward r_i, the *temporally extended reward function* $\bar{R}$ is defined as:

$$\bar{R}(\langle s_0, s_1, \dots, s_n \rangle) = \sum_{\phi_i \in \Phi : \langle s_0, s_1, \dots, s_n \rangle \models \phi_i} r_i \tag{4.15}$$

In order to run the actual learning task, (Bacchus et al., 1996) proposed a transformation from the NMRDP to an equivalent MDP with the state space *expaneded* which allows to label each state $s \in S$. The idea is that the labels should keep track in some way the (partial) satisfaction of the temporal formulas $\phi_i \in \Phi$. A state s in the transformed state space is replicated multiple times, marking the difference between different (relevant) histories terminating in state s.

In this way, they obtained a compact representation of the required history-dependent policy by considering only relevant history, and can produce this policy using computationally-effective MDP algorithms. In other words, the states of the NMRDP can be mapped into those of the expanded MDP, in such a way that corresponding states yield same transition probabilities and corresponding traces have same rewards.

4.5 RL for NMRDP with LTL$_f$/LDL$_f$ Rewards

In this section, we explain the main contribution of this chapter and one of the main contribution of the thesis. We devise a natural extension of the construction explained in (Brafman et al., 2018) for a reinforcement learning task. That is, we show that it is possible to do reinforcement learning for non-Markovian rewards, expressed in LTL$_f$/LDL$_f$ formulas, by applying an extension of the state space of the agent S, analogously to the one described in Section 4.4.4. In the first section, we describe the approach proposed by (Brafman et al., 2018); then, we observe that the expanded MDP can be used to do reinforcement learning to optimize LTL$_f$/LDL$_f$ non-Markovian rewards.

4.5.1 NMRDP with LTL$_f$/LDL$_f$ rewards

In this section, we explain how to specify non-Markovian rewards with LTL$_f$/LDL$_f$ formulas (instead of PLTL) and how the associated MDP expansion works (Brafman et al., 2018), analogously to what we saw with PLTL (Section 4.4.4).

The temporally extended reward function $\bar{R}$ is similar to Equation 4.15, but instead of using PLTL formula we use LTL$_f$/LDL$_f$ formulas. Formally, given a set of pairs $\{(\varphi_i, r_i)_{i=1}^m\}$ (where φ_i denotes the LTL$_f$/LDL$_f$ formula for specifying a desired behavior, and r_i denotes the reward associated to the satisfaction of φ_i, and given a (partial) trace $\pi = \langle s_0, s_1, \dots, s_n \rangle$, we define $\bar{R}$ as:

$$\bar{R}(\pi) = \sum_{1 \leq i \leq m : \pi \models \varphi_i} r_i \tag{4.16}$$

For the sake of clarity, in the following we use $\{(\varphi_i, r_i)_{i=1}^m\}$ to denote $\bar{R}$.

Now we describe the MDP expansion for doing learning in this setting, as proposed in (Brafman et al., 2018). Without loss of generality, we assume that every NMRDP $\mathcal{N}$ is reduced into another NMRDP $\mathcal{N}' = \langle S', A', T', R', \gamma\rangle$:

$$\begin{aligned} S' &= S \cup \{s_{init}\} \\ A' &= A \cup \{start\} \\ T'(s,a,s') &= \begin{cases} 1 & \text{if } \ s = s_{init}, a = start, s' = s_0 \\ 0 & \text{if } \ s = s_{init} \text{ and } (a \neq start \text{ or } s' \neq s_0) \\ T(s,a,s') & \text{otherwise} \end{cases} \\ R'(\langle s_{init}, s_0, \dots, s_n\rangle) &= R(\langle s_0, s_1, \dots, s_n\rangle) \end{aligned} \tag{4.17}$$

and s_{init} is the new initial state. In other words, we prefix to every feasible trajectory $\mathcal{N}$ the pair $\langle s_{init}, start\rangle$, denoting the beginning of the episode. We do this for two reasons: allow to evaluate formulas in s_0 and make it compliant with the most general definition of the reward, namely $R(s,a,s')$, also when there is no true action that is done (i.e. empty trace).

Definition 4.5 (Brafman et al. (2018))**.** Given an NMRDP $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m, \gamma\}\rangle$ (i.e. with non-Markovian rewards specified by LTL$_f$/LDL$_f$ formulas) it is possible to build an $\mathcal{M} = \langle S', A, T', R', \gamma\rangle$ that is *equivalent* (in the sense of Definition 4.3) to $\mathcal{N}$. Denoting with $\mathcal{A}_{\varphi_i} = \langle 2^{\mathcal{P}}, Q_i, q_{i0}, \delta_i, F_i\rangle$ (notice that $S \subseteq 2^{\mathcal{P}}$ and δ_i is total) the DFA associated with φ_i (see Section 2.5), the equivalent MDP $\mathcal{M}$ is built as follows:

- $S' = Q_1 \times \cdots \times Q_m \times S$ is the set of states;
- $T' : S' \times A \times S' \to [0,1]$ is defined as follows:

$$Tr'(q_1, \dots, q_m, s, a, q_1', \dots, q_m', s') = \\ \begin{cases} Tr(s,a,s') & \text{if } \forall i : \delta_i(q_i, s') = q_i' \\ 0 & \text{otherwise;} \end{cases}$$

- $R' : S' \times A \times S' \to \mathbb{R}$ is defined as:

$$R'(q_1, \dots, q_m, s, a, q_1', \dots, q_m', s') = \sum_{i : q_i' \in F_i} r_i$$

Theorem 4.2 (Brafman et al. (2018))**.** *The NMRDP $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)\}_{i=1}^m, \gamma\rangle$ is equivalent to the MDP $\mathcal{M} = \langle S', A, T', R', \gamma\rangle$ defined in Definition 4.5.*

Proof. Recall that every $s' \in S'$ has the form $(q_1, \dots, q_m, s)$. Define $\tau(q_1, \dots, q_m, s) = s$. Define $\sigma(s) = (q_{10}, \dots, q_{m0}, s)$. We have $\tau(\sigma(s)) = s$, hence Condition 1 is verified. Condition 2 of Definition 4.3 is easily verifiable by inspection. For Condition 3, consider a possible trace $\pi = \langle s_0, s_1, \dots, s_n\rangle$. We use σ to obtain $s_0' = \sigma(s_0)$ and given s_i, we define s_i' (for $1 \leq i \leq n$) to be the unique state $(q_{1,i}, \dots, q_{m,i}, s_i)$ such that $q_{j,i} = \delta(q_{j,i-1}, s_i)$ for all $1 \leq j \leq m$. Moreover, we require that, without loss of generality, every trajectory in the new MDP starts from s_{init} and now have a corresponding possible trace of $\mathcal{M}$, i.e.,

$\pi = \langle s'_0, s'_1, \ldots, s'_n \rangle$. This is the only feasible trajectory of $\mathcal{M}$ that satisfies Condition 3. The reward at $\pi = \langle s_0, s_1, \ldots, s_n \rangle$ depends only on whether or not each formula φ_i is satisfied by π. However, by construction of the automaton $\mathcal{A}_{\varphi_i}$ and the transition function T, $\pi \models \varphi_i$ iff $s'_n = (q_1, \ldots, q_m, s_n)$ and $q_i \in F_i$ □

Let ρ' be a (Markovian) policy for $\mathcal{M}$. It is easy to define an *corresponding* policy on $\mathcal{N}$, i.e., a policy that guarantees the same rewards, by using τ and σ mappings defined in Theorem 4.2 and the result shown in Theorem 4.4.

Obviously, typical learning techniques, such as Q-learning or Sarsa, are applicable on the expanded $\mathcal{M}$ and so we can learn an optimal policy ρ for $\mathcal{M}$. Thus, an optimal policy for $\mathcal{N}$ can be learnt on $\mathcal{M}$. Of course, none of these structures is (completely) known to the learning agent, and the above transformation is never done explicitly. Rather, the agent carries out the learning process by assuming that the underlying model is $\mathcal{M}$ instead of $\mathcal{N}$ (applying the fix introduced in Definition 4.17).

Observe that the state space of $\mathcal{M}'$ is the product of the state spaces of $\mathcal{N}$ and $\mathcal{A}_{\varphi_i}$, and that the reward R' is Markovian. In other words, the (stateful) structure of the LTL$_f$/LDL$_f$ formulas φ_i used in the (non-Markovian) reward of $\mathcal{N}$ is *compiled* into the states of $\mathcal{M}$.

Why should we use LDL$_f$

LDL$_f$ formalism (introduced in Section 2.4) has the advantage of *enhanced expressive power* over other proposals, as discussed in (Brafman et al., 2018). Indeed, we move from linear-time temporal logics to LDL$_f$, paying no additional (worst-case) complexity costs. LDL$_f$ can encode in polynomial time LTL$_f$, regular expressions (RE) and the past LTL (PLTL) of (Bacchus et al., 1996). Moreover, LDL$_f$ can naturally represent "procedural constraints" (Baier et al., 2008), i.e., sequencing constraints expressed as programs, using "if" and "while", hence allowing to express more complex properties.

4.5.2 RL for LTL$_f$/LDL$_f$ rewards

Now we made an important remark that will be used in the next chapter. That is, the transformation that led us from an NMRDP with LTL$_f$/LDL$_f$ rewards (Section 4.5.1) to an equivalent (*extended*) MDP allow us to do reinforcement learning to learn LTL$_f$/LDL$_f$ non-Markovian rewards.

In other words, we can do *reinforcement Learning for the NMRDP $\mathcal{N}$ with* LTL$_f$/LDL$_f$ *rewards* by simply reducing the problem to *reinforcement learning over the equivalent MDP $\mathcal{M}$*, where $\mathcal{M}$ is the result of the transformation applied to $\mathcal{N}$, as described before.

The actual learning is done over the expanded state space $S \times Q_1, \times \cdots \times Q_m$, where Q_i are the set of states of the DFA associated to the formula φ_i that define the non-Markovian rewards. An expanded state has the form $\mathbf{s} = \langle s, q_1, \ldots, q_m \rangle$. The $q_1, \ldots, q_m$ are the current state of the automaton that actually tracks the satisfaction of the associated formula φ_i, in the current episode, in an implicit and compact way. They are determined, during the learning process, by the observed $s \in S \subseteq 2^{\mathcal{P}}$ (look at how the transition function is defined in Definition 4.5).

This result has a practical importance since it does not require the definition of new RL algorithms and thus makes the learning process easy and effective.

Obviously, typical learning techniques, such as Q-learning or SARSA, are applicable on (the state space of) $\mathcal{M}$ and we can learn an optimal policy ρ for $\mathcal{M}$. Thus, an optimal policy for $\mathcal{N}$ can be learnt on $\mathcal{M}$. Of course, none of these structures is (completely) known to the learning agent, and the above transformation is never done explicitly. Rather, the agent carries out the learning process by assuming that the underlying model is $\mathcal{M}$ instead of $\mathcal{N}$. We remark that the state space of $\mathcal{M}$ is the product of the state spaces of $\mathcal{N}$ and $\mathcal{A}_{\varphi_i}$, and that the reward R is Markovian. In other words, the (stateful) structure of theLTL$_f$/LDL$_f$ formulas φ_i used in the (non-Markovian) reward of $\mathcal{N}$ is compiled into the states of $\mathcal{M}$.

We formally state the concepts just presented with the following theorem:

Theorem 4.3. *RL for* LTL$_f$/LDL$_f$ *rewards φ over an NMRDP $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)\}_{i=1}^m\rangle$, with T and r_i unknown to the learning agent, can be reduced to RL over the MDP $\mathcal{M}$ defined above.*

4.6 Conclusion

In this chapter, we introduced the topic of Reinforcement Learning, as well as foundational definitions (MDP, policy ρ, state-value function v_ρ) and algorithms (Q(λ), Sarsa(λ)). Then we presented the notion of NMRDP and a technique to build an equivalent MDP, by specifying non-Markovian reward function with LTL$_f$/LDL$_f$ formalisms. We end the chapter by remarking that the transformed MDP can be used for reinforcement learning, formalizing the results with a theorem.

Chapter 5

RL for LTL$_f$/LDL$_f$ Goals

In this chapter, we describe one of the main contributions of the thesis. We define a particular problem, that we call *Reinforcement Learning for* LTL$_f$/LDL$_f$ *Goals*, and propose a solution. We focus on the case where we have two separate representations of the world: one for the agent, using the *low-level* features available to it, and one for the goal, expressed in terms of *high-level* (human-understandable) fluents.

In the first section, we describe the context of the problem and the motivations, whereas in the second section we give a formal definition. In the next section, we describe the problem more formally. Then, we show how we can reduce this problem to do reinforcement learning on an equivalent MDP. Finally, we give an episodic view of the scenario, which is the one used in the implementation of the experiments.

5.1 Introduction and Motivations

Let consider to have a learning agent equipped with sensing procedures to compute a set of features from the world that forms its states and with a set of actions that it can do. For example, you can imagine a robot in a room that can move in many directions; its state space can be a combination of features measured from different physical quantities e.g. its position in the room.

Our purpose in this work is to use this agent to learn one (or simultaneously many) task whose goal (or many goals) are expressed in LTL$_f$/LDL$_f$. Such goals are expressed over a representation of the world that is not the one used by the agent (oversimplifying, we may say that the agent has a low-level representation), but a convenient high-level representation suitable to express declaratively temporally extended goals. In other words, we study the possibility of having *two separate representations of the world*:

- one for expressing the dynamics of the RL agent;
- one for expressing the LTL$_f$/LDL$_f$ goals.

These two representations use different classes of features from the real world: the first includes the features that the agent can directly access, while the second includes the features needed to evaluate the LTL$_f$/LDL$_f$ goal.

One may ask: why we need two *separated* representation of the world? Why can we not merge them into a unique representation and reason/learn over this joint model?

There are many reasons why this might be useful:

1. The agent feature space can be designed separately from the features needed to express the goal, thus promoting *separation of concerns* which, in turn, facilitates the design; this separation facilitates also the *reuse* of representations already available, possibility developed for the standard setting.

2. A reduced agent's feature space allows for realizing *simpler agents* while preserving the possibility of tackling complex declarative goals which cannot be represented in the agent's feature space.

3. Reducing the agent's feature space may yield a *reduced* state space to be explored by the RL-agent.

Clearly, the two separate representations (i.e., the two sets of features) need to be somehow correlated in reality. The crucial point, however, is that in order to perform RL effectively, *such a correlation does not need to be formalized.* We let the reinforcement learning agent learn, over its state space (eventually extended), the proper way to use an arbitrarily long sequence of low-level actions in order to accomplish the high-level goals.

Example 5.1. Consider the RL environment BREAKOUT (Example 4.2). Let the agent's state space be the combination of the paddle position, the ball position and the ball speed. The actions available to the agent are: move to the left, move to the right, and a nop-action. The goal is to remove all the bricks by hitting and directing the ball in a proper way.

Now suppose that we want to express a LTL$_f$/LDL$_f$ goal, which is a sort refinement of the agent's goal: to break columns of bricks in a precise order (e.g. from left to right). To express this goal we need:

- a set of features that allow detecting the status of the bricks;
- a mapping of those features to a set of fluents upon which the LTL$_f$/LDL$_f$ formula can be expressed (e.g. the status of each column of bricks).

In this case, the agent's state space represents the *low-level* state space, whereas all the possible configurations of the fluents represent the *high-level* state space.

Notice that for the satisfaction of the LTL$_f$/LDL$_f$ goal defined above, we don't need to include every possible configuration of the fluents (i.e. status of every brick) into the state space of the learning agent; instead, our intuition suggests that the RL agent, in order to be able to learn the optimal action in a given state, *needs only to keep track at which point of the satisfaction of the goal it is*, such that it is able to learn how to proceed, until the complete satisfaction of the goal.

In our example, the learning agent just needs to know which is the next column of bricks to remove. Hence, besides the agent's state space, the agent needs to know the current target column, up to n different sub-goals, where n is the number of columns. In this way, it can learn how to break the current target column, and inherently the sequence of actions that lead to the removing of every column, one by one, in sequence.

Another question may arise: why one should need the proposed construction (i.e. the extended state space for temporal goals from the LTL$_f$/LDL$_f$ specification) instead of a state space composed by ad-hoc features set for specifying the desired temporal goal?

The last considerations and the following ones should convince the reader about the main advantages of our approach:

- *Reduced state space* (motivation 3). Let S be the agent's state space. The minimum size of the learning state space needed to the agent is $|S| \cdot n$, where the n-fold increase of S allow to discern a state $s \in S$ in different partial satisfaction of the goal (i.e. the last column removed). Hence, we do not need to include directly the status of each brick, which would return a feature space of size $|S| \cdot 2^{mn}$ (n i the number of columns of bricks and m is the number of rows). Notice that the latter state space is *exponentially* bigger than the minimal one, which is only linear in the number of columns.

- *Easier feature design* (motivation 1). Even if, with traditional feature engineering, it is possible to build a composed state space that includes the needed pieces of information for learning temporally extended goal, the result is a hard-coded model, rigid to modifications. On the other hand, our approach is more flexible and open to modifications, both on features/fluents level and on LTL$_f$/LDL$_f$ goal level (just add/modify LTL$_f$/LDL$_f$ formulas).

- *Declarativeness*: specifying by hand how the state components that track the temporal goal should evolve could be very hard and unintuitive. Instead, LTL$_f$/LDL$_f$ formalisms allow to easily express a wide range of temporal goals about how fluents should change over time.

How the learning state space (i.e. the state space over where the agent actually learns) can be deduced by the LTL$_f$/LDL$_f$ formula and the fluents configuration is the topic of the following sections.

In order to proceed with a formal definition of the problem, we made the following working assumption:

- The agent's model of the world is an MDP; in particular, the transitions from a state to another state, given an action, satisfies the Markov property;

- The granularity of the state representations is enough to accomplish the LTL$_f$/LDL$_f$ goals. It might be the case that the high-level or the low-level representation of the world is somehow inadequate to capture the relevant changes in the world;

- Notice that, in our setting, from the agent's point of view, the changes of the agent's state space and the ones of the fluents are *synchronous*, and depends from a *clock*, implicit in the model definition. Hence, we assume that this synchrony, as well as the granularity resulting from the clock, still allow the agent to learn to properly optimize the rewards.

5.2 Problem definition

In this section, we describe more formally the scenario. We first give an overall picture of the model. then, we make an important observation about the transition function of the joint world representations. Finally, we define the mathematical model that will be used in the next section.

5.2.1 The World, the Agent and the Fluents

Let $\mathcal{W}$ be a *world* of interest (e.g. a room, an environment, a video game). Let W be the set of *world states*, i.e. the states of the world $\mathcal{W}$. A *feature* is a function f_j that maps a world state to the values of another domain D_j, such as reals, finite enumerations, booleans, etc., i.e., $f_j : W \to D_j$. Given a set of features $F = \langle f_1, \ldots, f_d \rangle$, the *feature vector* of a world state w_h is the vector $\mathbf{f}(w_h) = \langle f_1(w_h), \ldots, f_d(w_h) \rangle$ of feature values corresponding to w_h.

Now consider an agent that lives in $\mathcal{W}$. The agent can interact with $\mathcal{W}$ by executing an action a taken from a *set of actions* A. Without loss of generality, we assume that such a learning agent has a special action *stop* which deems the end of an episode. Moreover, the agent has its own set of features $F_{ag} = \langle f_1, \ldots, f_d \rangle$, which yields its *representation of the world* S, where $S \subseteq F_{ag}(W)$ and $F_{ag}(W) = \{\mathbf{f}_{ag}(w) | w \in W\}$. We assume that the agent has a *clock* which determines the granularity of its acting. At every clock, the agent can do action a and observe both the new state s from the new world state w', namely $s = \mathbf{f}_{ag}(w)$, and a real-valued reward $R(s, a, s')$, that depends only from the transition $s \to_a s'$. Finally, we assume that the law that determines the possible next state s' given the history of a trajectory (i.e. the state transition function T) is Markovian, hence it depends only from the current state s and the taken action a. In other words, we can define an MDP $\mathcal{M}_{ag} = \langle S, A, T, R, \gamma \rangle$.

We consider arbitrarily LTL$_f$/LDL$_f$ formulas φ_i $(i = 1, \ldots, m)$ over a set of fluents $\mathcal{F}$ used to provide a high-level description of the world. We denote by $\mathcal{L} = 2^{\mathcal{F}}$ the set of possible fluents configurations. Given a set of features F_{goal}, a *configuration of fluents* $\ell_h \in \mathcal{L}$ is formed by the components that assign truth values to the fluents according to the feature vector $\mathbf{f}_{goal}(w_h)$. At every step, the features for fluents evaluations are observed, obtaining a particular configuration $\ell \in \mathcal{L}$. Notice that in general the features for the fluents and for the agent state space may differ. The formula φ_i is selecting sequences of fluents configurations $\ell_1, \cdots, \ell_n$, with $\ell_k \in \mathcal{L}$, whose relationship with the sequences of states $s_1, \ldots, s_n$, with $s_k \in S$ is unknown.

In other words, a subset of features are used to describe agent states s_h and another subset (for simplicity, assumed disjoint from the previous one) are used to evaluate the fluents in ℓ_h. Hence, given a sequence $w_1, \ldots, w_n$ of world states we get the corresponding sequence of sequences learning agent states $s_1, \ldots, s_n$ and simultaneously the sequence of fluent configurations $\ell_1, \ldots, \ell_n$. Notice that we do not have a formalization for $w_1, \ldots, w_n$ but we do have that for $s_1, \ldots, s_n$ and for $\ell_1, \ldots, \ell_n$.

Oversimplifying, we may say that S is the set of configurations of the low-level features for the learning agent, while $\mathcal{L}$ is the set of configuration of the high-level features needed for expressing φ_i.

Figure 5.1 depicts the above described scenario. You can recognize how the world state w contributes both to the generation of the agent's state s and of the fluents configurations ℓ.

5.2.2 Joint transition function

Now we consider the transition distribution over the features, the agent actions in A and the fluents configuration, i.e.,

$$T^g_{ag} : S \times \mathcal{L} \times A \to Prob(S \times \mathcal{L}) \tag{5.1}$$

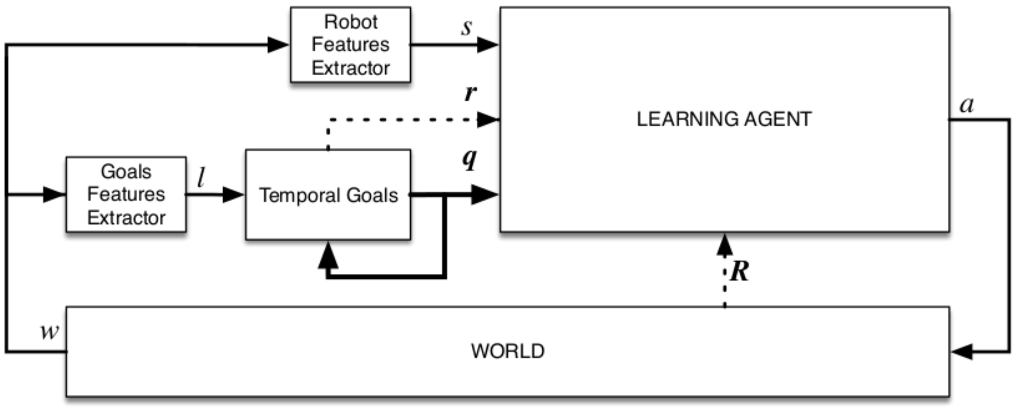


Figure 5.1. The reinforcement learning system for temporally extended goals.

We observe that the state transition function T^g_{ag} is *Markovian*, i.e. the probability to end in the next state s' with the next fluents configuration ℓ' depends only from s, ℓ and a (the current agent state, the current fluents configuration and the action taken, respectively).

Actually, the observation is necessary, under the assumption that the evolution of the agent's state space component is Markovian. Indeed, it might happen that the introduction of the fluents configurations in the transition model violates the Markov property of the transition function in Equation 5.1; however, we notice that, if this is the case, we can expand $\mathcal{L}$ by including many other fluents, as much as it is needed to make the transition function Markovian.

We will see that this mathematical trick does not affect the following results, because $\mathcal{L}$, at the end of the day, *does not matter to the learning agent*. So, without loss of generality, we consider the joint transition function in Equation 5.1 is Markovian, even if the considered set of fluents configurations $\mathcal{L}$ makes the transition function to violates the Markov property for some transitions.

Such a transition distribution together with the initial values of the fluents ℓ_0 and of the agent state s_0 allow us to describe a probabilistic transition system accounting for the dynamics of the fluents and agent states. In other words, in response to an agent action a_h performed in the current state w_h (in the state s_h of the agent and the configuration ℓ_h of the fluents), the world changes into w_{h+1} from which s_{h+1} and ℓ_{h+1}. This is all we need to proceed.

5.2.3 Formal definition

We are interested in devising policies for the learning agent such that at the end of the episode, i.e., when the agent executes *stop*, the LTL$_f$/LDL$_f$ goal formulas φ_i $(i = 1, \ldots, m)$ are satisfied. Now we can state our problem formally.

Definition 5.1. *We define* RL for LTL$_f$/LDL$_f$ goals*, denoted as*

$$\mathcal{M}_{ag}^{goal} = \langle S, A, R, \mathcal{L}, T_{ag}^g, \{(\varphi_i, r_i)\}_{i=1}^m \rangle$$

with T_{ag}^g, R and r_i unknown,the following problem: given a learning agent $\mathcal{M}_{ag} = \langle S, A, T, R \rangle$, with T and R unknown and a set $\{(\varphi_i, r_i)\}_{i=1}^m$ of LTL$_f$/LDL$_f$ *formulas with associated rewards, find a (non-Markovian) policy $\bar{\rho} : S^* \to A$ that is optimal wrt the sum of the rewards r_i and R.*

Observe that an optimal policy for our problem, although not depending on $\mathcal{L}$, is guaranteed to satisfy the LTL$_f$/LDL$_f$ goal formulas.

5.3 Reduction to MDP

To devise a solution technique, we start by transforming $\mathcal{M}_{ag}^{goal} = \langle S, A, T_{ag}^g, R, \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^m \rangle$ into an NMRDP $\mathcal{M}_{ag}^{nmr} = \langle S \times \mathcal{L}, A, T_{ag}^g, \{(\varphi_i', r_i)\}_{i=1}^m \cup \{(\varphi_s, R(s,a,s'))\}_{s \in S, a \in A, s' \in S} \rangle$ where:

- States are pairs (s, ℓ) formed by an agent configuration s and a fluents configuration ℓ.
- $\varphi_i' = \varphi_i \wedge \Diamond Done$.
- $\varphi_s = \Diamond(s \wedge a \wedge \circ(Last \wedge s'))$.
- T_{ag}^g, r_i and $R(s,a,s')$ are unknown and sampled from the environment.

Formulas φ_i' simply require to evaluate the corresponding goal formula φ_i after having done the action *stop*, which sets the fluent *Done* to true and ends the episode. Hence it gives the reward associated with the goal at the end of the episode. The formulas $\Diamond(s \wedge a \wedge \circ(Last \wedge s'))$, one per (s, a, s'), requires both states s and action a are followed by s' are evaluated at the end of the current (partial) trace (notice the use of *Last*). In this case, the reward $R(s,a,s')$ from $\mathcal{M}_{ag}$ associated with (s,a,s') is given.

Notice that policies for $\mathcal{M}_{ag}^{nmr}$ have the form $(S \times \mathcal{L})^* \to A$ which needs to be restricted to have the form required by our problem $\mathcal{M}_{ag}^{goal}$.

A policy $\bar{\rho} : (S \times \mathcal{L})^* \to A$ *has the form* $S^* \to A$ when for any sequence of n states $\langle s_1 \cdots s_n \rangle$, we have that for any pair of sequences of fluent configurations $\langle \ell_1' \cdots \ell_n' \rangle$, $\langle \ell_1'' \cdots \ell_n'' \rangle$ the policy returns the same action, $\bar{\rho}(\langle s_1, \ell_1' \rangle \cdots \langle s_n, \ell_n' \rangle) = \bar{\rho}(\langle s_1, \ell_1'' \rangle \cdots \langle s_n, \ell_n'' \rangle)$. In other words, a policy $\bar{\rho} : (S \times \mathcal{L})^* \to A$ has the form $\bar{\rho} : S^* \to A$ when it does not depend on the fluents $\mathcal{L}$. We can now state the following result.

Theorem 5.1. *RL for* LTL$_f$/LDL$_f$ *goals $\mathcal{M}_{ag}^{goal} = \langle S, A, Tr_{ag}^g, R, \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^m \rangle$ can be reduced to RL over the NRMDP $\mathcal{M}_{ag}^{nmr} = \langle S \times \mathcal{L}, A, T_{ag}^g, \{(\varphi_i', r_i)\}_{i=1}^m \cup \{(\varphi_s, R(s,a,s'))\}_{s \in S, a \in A, s' \in S} \rangle$, restricting policies to be learned to have the form $S^* \to A$.*

Observe that by restricting $\mathcal{M}_{ag}^{nmr}$ policies to S^*, in general, we may discard policies that have a better reward but depend on $\mathcal{L}$. On the other hand, these policies need to change the learning agent in order to allow it to observe $\mathcal{L}$ as well. As mentioned in the introduction, we are interested in keeping the learning agent as it is, apart from additional memory.

As a second step, we apply the construction of Section 4.5 and obtain a new MDP learning agent. In such construction, however, because of the triviality of their automata, we do not need to keep track of the state φ_s, but just give the reward $R(s, a, s')$ associated to (s, a, s'). Instead, we do need to keep track of the state of the DFAs $\mathcal{A}_{\varphi_i}$ corresponding to the formulas φ'_i. Hence, from $\mathcal{M}_{ag}^{nmr}$, we get an MDP $\mathcal{M}'_{ag} = \langle S', A', Tr'_{ag}, R' \rangle$ where:

- $S' = Q_1 \times \cdots \times Q_m \times S \times \mathcal{L}$ is the set of states;
- $Tr'_{ag} : S' \times A' \times S' \to [0, 1]$ is defined as follows:

$$Tr'_{ag}(q_1, \dots, q_m, s, \ell, a, q'_1, \dots, q'_m, s', \ell') = \\ \begin{cases} Tr(s, \ell, , a, s', \ell') & \text{if } \forall i : \delta_i(q_i, \ell') = q'_i \\ 0 & \text{otherwise;} \end{cases}$$

- $R' : S' \times A \times S' \to \mathbb{R}$ is defined as:

$$R'(q_1, \dots, q_m, s, \ell, a, q'_1, \dots, q'_m, s', \ell') = \\ \textstyle\sum_{i:q'_i \in F_i} r_i + R(s, a, s')$$

Finally we observe that the environment gives now both the rewards $R(s, a, s')$ of the original learning agent, and the rewards r_i associated to the formula so has to guide the agent towards the satisfaction of the goal (progressing correctly the DFAs $\mathcal{A}_{\varphi_i}$).

By applying Theorem 4.2 we get that NMRDP $\mathcal{M}_{ag}^{nmr}$ and the MDP $\mathcal{M}'_{ag}$ are equivalent, i.e., any policy of $\mathcal{M}_{ag}^{nmr}$ has an equivalent policy (hence guaranteeing the same reward) in $\mathcal{M}'_{ag}$ and vice versa. Hence we can learn policy on $\mathcal{M}'_{ag}$ instead of $\mathcal{M}_{ag}^{nmr}$.

We can refine Theorem 5.1 into the following one.

Theorem 5.2. *RL for* LTL$_f$/LDL$_f$ *goals $\mathcal{M}_{ag}^{goal} = \langle S, A, T_{ag}^{g}, R, \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^{m} \rangle$ can be reduced to RL over the MDP $\mathcal{M}'_{ag} = \langle S', A, T'_{ag}, R' \rangle$, restricting policies to be learned to have the form $Q_1 \times \ldots \times Q_n \times S \to A$.*

As before, a policy $Q_1 \times \ldots \times Q_n \times S \times \mathcal{L} \to A$ *has the form* $Q_1 \times \ldots \times Q_n \times S \to A$ when any ℓ and ℓ' the policy returns the same action, $\rho(q_1, \dots, q_n s, \ell) = \rho(q_1, \dots, q_n s, \ell')$.

The final step is to solve our original RL task on $\mathcal{M}_{ag}^{goal}$ by performing RL on a new MDP $\mathcal{M}_{ag}^{new} = \langle Q_1 \times \cdots \times Q_m \times S, A, T''_{ag}, R'' \rangle$ where:

- Transitions distribution T''_{ag} is the marginalization wrt $\mathcal{L}$ of T'_{ag} and is unknown;
- Rewards R'' is defined as:

$$R''(q_1, \dots, q_m, s, a, q'_1, \dots, q'_m, s') = \sum_{i:q'_i \in F_i} r_i + R(s, a, s').$$

- States q_i of DFAs $\mathcal{A}_{\varphi_i}$ are progressed correctly by the environment.

Indeed, we can show the following result.

Theorem 5.3. *RL for* LTL$_f$/LDL$_f$ *goals $\mathcal{M}_{ag}^{goal} = \langle S, A, T', R, \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^{m} \rangle$ can be reduced to RL over the MDP $\mathcal{M}_{ag}^{new} = \langle Q_1 \times \cdots \times Q_m \times S, A, T''_{ag}, R'' \rangle$ and the optimal policy ρ_{ag}^{new} learned for $\mathcal{M}_{ag}^{new}$ can be reduced to a corresponding optimal policy for $\mathcal{M}_{ag}^{goal}$.*

Proof. From Theorem 5.2, by the following observations. For the sake of brevity, we use $\mathbf{q}$ to denote $q_1, \dots, q_m$. Notice also that for all $\ell, \ell' \in \mathcal{L}$, $R'(\mathbf{q}, s, \ell, a, \mathbf{q}', s', \ell') = R''(\mathbf{q}, s, a, \mathbf{q}', s')$.

We show that the values of $v^\rho_{ag}(q_1, \dots, q_m, s, \ell)$, i.e. the state value function for $\mathcal{M}'_{ag}$(for simplicity v^ρ, unless otherwise stated), for some policy ρ, do not depend on ℓ or, in other words, it is necessary that $\forall \ell_1, \ell_2 . v^\rho(q_1, \dots, q_m, s, \ell_1) = v^\rho(q_1, \dots, q_m, s, \ell_2)$. Finally, we notice that $\forall \ell . v^{\rho,new}_{ag} = v^\rho_{ag}$

From Equation 4.4 we have:

$$\begin{gathered} v_\rho(\mathbf{q}, s, \ell) = \\ \sum_{\mathbf{q}', s', \ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, \ell, a)[R'(\mathbf{q}, s, \ell, a, \mathbf{q}', s', \ell') + \gamma v_\rho(\mathbf{q}', s', \ell')] = \\ \sum_{\mathbf{q}', s', \ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, \ell, a)[R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v_\rho(\mathbf{q}', s', \ell')] \end{gathered} \tag{5.2}$$

Using the equivalence between R' and R'', as already pointed out. Notice that we can compute $\mathbf{q}'$ from $\mathbf{q}$ and ℓ', hence we do not need ℓ. In other words:

$$P(\mathbf{q}', s', \ell' | \mathbf{q}, s, \ell, a) = P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)$$

Equation 5.2 becomes:

$$\sum_{\mathbf{q}', s', \ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)[R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v_\rho(\mathbf{q}', s', \ell')] \tag{5.3}$$

At this point, we see that v^ρ does not depend from ℓ, hence we can safely drop ℓ as argument for v_ρ, obtaining v^ρ_{ag}. Indeed, from 5.3:

$$\begin{gathered} \sum_{\mathbf{q}', s'} [R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v^{\rho,new}_{ag}(\mathbf{q}, s)] \sum_{\ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a) = \\ \sum_{\mathbf{q}', s'} P(\mathbf{q}', s' | \mathbf{q}, s, a)[R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v^\rho_{ag}(\mathbf{q}', s')] = \\ v^\rho_{ag}(\mathbf{q}, s) \end{gathered} \tag{5.4}$$

Where in 5.4 we marginalized the distribution $P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)$ over ℓ'. From Definition 4.1 of optimal policy, we can reduce an optimal policy ρ^{new}_{ag} to a policy of the form $\rho'_{ag} : Q_1 \times \dots \times Q_m \times S \to A$ that is optimal for $\mathcal{M}'_{ag}$ (since the state value function of $\mathcal{M}^{new}_{ag}$, after dropping the argument ℓ, and of $\mathcal{M}'_{ag}$ are equivalent). From Theorem 5.2 the thesis. □

It is worth to remark that in the resulting MDP $\mathcal{M}^{new}_{ag}$ the *explicit presence of the fluents configuration* ℓ *has been removed.* Rather, the dependency is compiled into the expanded state space $Q_1 \times \dots Q_m \times S$, where $Q_1, \dots, Q_m$ are the automata state spaces associated to the formulas φ_i.

5.4 An episodic goal-based view

Here we clarify how the actual transition model works in the final MDP $\mathcal{M}_{ag}^{new}$. We focus on the episodic view, where the learning process is organized in episodes. Recall that the state space of $\mathcal{M}_{ag}^{new}$ is $Q_1 \times \dots Q_m \times S$, where Q_i is the set of states of $\mathcal{A}_{\varphi_i}$, the automaton associated to the $\text{LTL}_f/\text{LDL}_f$ formula φ_i (see Section 2.5). Moreover we consider two cases: when there exists a subset of states $S_{goal} \subseteq S$ that we call *goal states*, such that every transition in those state make the task completed and hence the end of the episode; and when there are no goal states, but the task is to maximize the obtained reward, until a maximum number of steps is reached. E.g. in Gridworld presented in Example 4.1, we can defined s_{34} as a goal state. However, nothing prevents us to set a maximum number of time steps, and let the agent learn how to collect reward as much as possible in a limited amount of time.

Now we give the following definition:

Definition 5.2. Let $\mathcal{M}_{ag}^{goal}$ be our problem and $\mathcal{M}_{ag}^{new}$ its transformation into and MDP, as defined in Theorem 5.3 and let $\mathbf{s} = \langle q_1, \dots, q_m, s\rangle \in Q_1 \times Q_1 \times \dots Q_m \times S$. Given an observation $s' \in S$ and $\ell' \in \mathcal{L}$, we define the *successor state of* $\mathbf{s}$ as $\mathbf{s}' = \langle q_1', \dots, q_m', s'\rangle$, where $q_i' = \delta_i(q_i, \ell')$.

A reinforcement learning episode in this setting works as follows. Assume that the MDP has a dummy initial state s_{init} and a dummy action *start*, as defined for NMRDPs in 4.17. In this construction, the dummy initial state in the expanded state space is $(q_{0,0}, q_{1,0}, \dots, q_{m,0}, s_{init})$. The first action taken by the agent is *start*, which allow the agent to observe the true initial world state w_0. From w_0, the agent extract the features to determine the state $s_0 \in S$ and the fluents configuration $\ell_0 \in \mathcal{L}$ [1]. The successor state is $(q_0', q_1', \dots, q_m', s_0)$. Then the agent might take a new action, observe another world state w', extract $s' \in S$ and $\ell' \in \mathcal{L}$, and compute the q_i as before. The sequence reiterates until the end of the episode.

Consider a generic state $\mathbf{s} = \langle q_1, \dots, q_m, s\rangle \in Q_1 \times Q_1 \times \dots Q_m \times S$ and a transition to $\mathbf{s}' = \langle q_1', \dots, q_m', s'\rangle$. For each new state q_i' of automaton $\mathcal{A}_{\varphi_i}$, the following might happen:

1. $q_i' = q_i$, i.e. the state is not changed;
2. $q_i' \neq q_i$, and from q_i' it is still possible to reach a final state;
3. $q_i' \neq q_i$, and from q_i' any final state cannot be reached;

If, for some $\mathcal{A}_{\varphi_i}$, we are in case 3, then the constraint specified by φ_i is violated, hence we call $\mathbf{s}'$ a *failure state*. We say that $\mathbf{s}$ is a *goal state* if $\forall i. q_i \in F_i$, i.e. every current state q_i is in an accepting state of the automaton $\mathcal{A}_{\varphi_i}$. Instead, if the underlying MDP is a goal-based one (e.g. $S_{goal} \neq \emptyset$) then $\mathbf{s}$ is a goal state if $s \in S_{goal}$.

Let $\mathbf{s}$ the last state of the trace of the current episode. The *stopping condition*, i.e. the condition that determines the end of the episode, depending from $\mathbf{s}$, is:

$$\text{failure\_state}(\mathbf{s}) \vee \text{goal\_state}(\mathbf{s}) \vee \text{exceeded\_time\_limit}$$

[1]Observe that the first transition is "artificial", and in many cases, both s_0 and ℓ_0 are known to the experimenter. The explanation presented here is aimed to clarify the underlying mathematical construction.

The reward $R(s, a, s')$ is collected after each taken action, and it is summed with r_i for every satisfied φ_i at the last state of the episode.

Remark about the stopping condition: notice that the presence of failure_state($\mathbf{s}$) in the stopping condition is not strictly needed. Indeed, in the general case, the agent still tries to learn an optimal policy in terms of observed rewards, regardless of the temporal goal at some point of the episode cannot be satisfied anymore (i.e. a failure state for some automaton $\mathcal{A}_{\varphi_i}$ is reached). However, we can think of failure_state($\mathbf{s}$) as a way to avoid parts of simulations that are not of interest, since the trajectory is "compromised" in terms of satisfaction of LTL$_f$/LDL$_f$ formulas.

Summary

In the following we summarize the results of this Chapter:

- We defined a new problem: *Reinforcement Learning for* LTL$_f$/LDL$_f$ *Goals* $\mathcal{M}_{ag}^{goal}$ (Definition 5.1). In a nutshell, from an existing MDP we introduced temporal goals defined by LTL$_f$/LDL$_f$ formulas about fluents $\mathcal{L}$ observed from the world. A solution for this problem is a (non-Markovian) policy that is optimal in terms of rewards and such that satisfies the LTL$_f$/LDL$_f$ specifications.

- We gave an equivalent formulation of $\mathcal{M}_{ag}^{goal}$, the NMRDP $\mathcal{M}_{ag}^{nmr}$, and observe that the original problem can be reduced to this formulation (Theorem 5.1).

- We applied the construction shown in Section 4.5, yielding $\mathcal{M}'_{ag}$, and by using Theorem 4.2 we stated Theorem 5.2, showing that $\mathcal{M}_{ag}^{goal}$ can be reduced to $\mathcal{M}'_{ag}$

- Finally, by proving Theorem 5.3, we showed that we can do reinforcement learning over the equivalent MDP $\mathcal{M}_{ag}^{new}$ by simply dropping the fluents ℓ from the state space. The optimal policy for $\mathcal{M}_{ag}^{new}$ can be transformed to a solution for the original problem $\mathcal{M}_{ag}^{goal}$.

5.5 Conclusion

In this chapter, we defined and studied a new problem, *Reinforcement Learning for* LTL$_f$/LDL$_f$ *goals*, and proposed a reduction that yields an equivalent MDP which can be used to solve the original problem. We first informally introduced the novelty and the motivations of our construction. Then we defined the problem more formally. After that, we described the technical details of the reduction, by leveraging results and concepts from Chapter 4. Finally, we gave some details to specify an episodic reinforcement learning view of the built model.

Chapter 6

Automata-based Reward shaping

In this chapter, we discuss a method to improve exploration of the state space and improve the convergence rate in the setting studied in Chapters 4 and 5, in particular in the construction $\mathcal{M}_{ag}^{new}$. Indeed, the state space of the original MDP $\mathcal{M}_{ag}$ is expanded in order to implicitly label the states with relevant histories for the satisfaction of LTL$_f$/LDL$_f$ formulas, which in general makes harder to learn an optimal policy in $\mathcal{M}_{ag}^{new}$ wrt $\mathcal{M}_{ag}$, due to a bigger state space. Moreover, the introduction of temporal goals makes things harder, because the agent has to find a proper behaviour that satisfies all the goals. In general, proper behaviours are harder to find and require more exploration of the state space.

Reward Shaping is a general method, well-known in the literature of Reinforcement Learning, used to deal with big state spaces and *sparse* rewards, and trying to address the temporal credit assignment problem, i.e. to determine the long-term consequences of actions. It aims to help the agent by giving extra reward signals, besides the ones that are given by the environment, to guide the agent towards higher environment rewards. It is a way to implicitly include prior knowledge in a reinforcement learning setting and doing so the state space is explored more efficiently, as well as the convergence rate is sped-up.

In this chapter, we propose a technique to apply reward shaping in our setting. In particular, we devise two methods: one where the automaton is known apriori, that we call *off-line* reward shaping, and the other where the automaton is built from scratch, that we call *on-the-fly* reward shaping.

The chapter is structured as follows: in the first section, we explain the reward shaping theory and, in particular, the requirements for theoretical guarantees of policy invariance under reward transformation. Then we show how to apply reward shaping on the automata transitions associated to LTL$_f$/LDL$_f$ formulas φ_i, both in *off-line* variant (i.e. when the automaton is built *before* the beginning of the learning process) and *on-the-fly* variant (i.e. when the automaton is built *during* the learning process).

6.1 Reward Shaping Theory

Reward shaping is a well-known technique to guide the agent during the learning process and so reduce the time needed to learn. The idea is to supply additional rewards in a proper manner such that the optimal policy is the same as the original MDP.

More formally, consider as example the temporal difference in SARSA after a transition $s \rightarrow_a s'$, presented in Equation 4.10:

$$\delta = R(s, a, s') + \gamma Q(s', a') - Q(s, a) \tag{6.1}$$

Reward shaping consists in defining the *shaping function* $F(s, a, s')$ and sum it to the environment reward $R(s, a, s')$, namely:

$$\delta = R(s, a, s') + F(s, a, s') + \gamma Q(s', a') - Q(s, a) \tag{6.2}$$

In the following sections, we will discuss two way to define $F(s, a, s')$.

6.1.1 Potential-Based Reward Shaping (PBRS)

We give the definition of *potential-based shaping function.*

Definition 6.1 (Ng et al. (1999))**.** Let any S, A, γ and any shaping function $F : S \times A \rightarrow \mathbb{R}$ be given. We say F is a potential-based shaping function if there exists a real-valued function $\Phi : S \rightarrow \mathbb{R}$ such that for all $s \in S, a \in A, s' \in S$

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{6.3}$$

Notice that in Equation 6.1 the action does not affect the value of $F(s, a, s')$, hence sometime we write $F(s, s')$.

In (Ng et al., 1999) it has been shown the following theorem:

Theorem 6.1 (Ng et al. (1999))**.** *Given an MDP $\mathcal{M} = \langle S, A, T, R, \gamma\rangle$ and a potential based shaping function F (Definition 6.1). Then, consider the MDP $\mathcal{M}' = \langle S, A, T, R + F, \gamma\rangle$, i.e. the same of $\mathcal{M}$ but applying reward shaping. Then, the fact that F is a potential-based reward shaping function is a necessary and sufficient condition to guarantee consistency with the optimal policy. In particular:*

- *(Sufficiency) if F is a potential-based shaping function, then every optimal policy in $\mathcal{M}'$ is optimal in $\mathcal{M}$.*
- *(Necessity) if F is not a potential-based shaping function (e.g. no such Φ exists satisfying 6.3), then there exists T and R such that no optimal policy in $\mathcal{M}'$ is optimal in $\mathcal{M}$.*

In poor words, potential-based reward shaping of the form $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$, for some $\Phi : S \rightarrow \mathbb{R}$, is a necessary and sufficient condition for policy invariance under this kind of reward transformation, i.e. the optimal and near-optimal solutions of $\mathcal{M}$ are preserved when considering $\mathcal{M}'$.

6.1.2 Dynamic Potential-Based Reward Shaping (DPBRS)

A limitation of PBRS is that the potential of a state does not change dynamically during the learning. This assumption often is broken, especially if the reward-shaping function is generated automatically.

Equation 6.3 can be extended to include also the time as parameter of the potential function Φ, while guaranteeing policy invariance. Formally:

$$F(s, t, a, s', t') = \gamma\Phi(s', t') - \Phi(s, t) \tag{6.4}$$

where t and t' are respectively the time when visiting s and s'. In this case, we call this technique *dynamic potential-based reward shaping* (DPBRS).

The shaping function in the form 6.4 guarantees policy invariance, as shown in Theorem 6.1. To show why this is the case, consider the expected discounted return for an infinite sequence of states (Definition 4.1):

$$G = \sum_{k=0}^{\infty} \gamma^k R_k \tag{6.5}$$

If we apply dynamic potential-based reward shaping to Equation 6.5 we have:

$$\begin{aligned}
G_\Phi &= \sum_{k=0}^{\infty} \gamma^k (R_k + F(s_k, t_k, s_{k+1}, t_{k+1})) \\
&= \sum_{k=0}^{\infty} \gamma^k (R_k + \gamma\Phi(s_{k+1}, t_{k+1}) - \Phi(s_k, t_k)) \\
&= \sum_{k=0}^{\infty} \gamma^k R_k + \sum_{k=0}^{\infty} \gamma\Phi(s_{k+1}, t_{k+1}) - \sum_{k=0}^{\infty} \Phi(s_k, t_k) \\
&= G + \sum_{k=1}^{\infty} \gamma\Phi(s_k, t_k) - \sum_{k=1}^{\infty} \Phi(s_k, t_k)) - \Phi(s_0, t_0) \\
&= G - \Phi(s_0, t_0)
\end{aligned}$$

Hence, any expected reward with reward shaping is the same of the one without reward shaping but a negative shift equal to $\Phi(s_0, t_0)$, i.e. a constant that does not depend from the actions taken. This means that the policy cannot be affected by the shaping function.

6.1.3 Relevant considerations about PBRS

Here we talk about some issues in PBRS described in Section 6.1.1 (analogous considerations can be made for DPBRS described in Section 6.1.2), described in (Grzes and Kudenko, 2009; Grzes, 2010; Grześ, 2017). In particular, we focus on:

- the presence of the discount factor γ in Equation 6.3;
- the value of $\Phi(s)$ when s is a terminal state.

The discount factor γ

In order to guarantee policy invariance, γ in Equation 6.3 must be equal to the discount factor of the MDP. Observe that, in general, this does not imply a *speed-up in learning* *time*. Indeed in (Grzes, 2010; Grzes and Kudenko, 2009) several issues of PBRS have been described, when $\gamma < 1$, that worsen the learning. In particular, it might happen that for some chosen value of $\Phi(s)$, the shaping function does not give a meaningful reward signal, e.g. near to the goal state, instead of a positive reward, a negative one signal is given to the learner, which is obviously a counterproductive choice.

In (Grzes, 2010) has been proposed an alternative approach to PBRS, which simply sets $\gamma = 1$ in Equation 6.3, namely:

$$F(s, a, s') = \Phi(s') - \Phi(s) \tag{6.6}$$

It has be proven that this approach *does not guarantee policy invariance*, i.e. the policy learned over the MDP with reward shaping in general is not equivalent to the original MDP. In the same work, it has been shown experimental evidence of the goodness of the new approach, even in the pathological cases with $\gamma < 1$. So using PBRS with $\gamma_{rs} = 1$, even if the discount factor of the MDP $\gamma_{mdp} \neq 1$, "works", although the invariance of the policy is not guaranteed anymore.

The value of terminal state $\Phi(s)$

In (Grześ, 2017) has been explained that the potential function in any terminal state (i.e. in any state where the episode terminates), must be 0 in order to guarantee policy invariance.

In Figure 6.1 is depicted a particular scenario in which the violation of this requirement over potential-based reward shaped learning leads to a different policy than non-shaped learning. In particular, without reward shaping the optimal policy from s_i would choose g_2 instead of g_1, since $r_{g_2} = 100 > r_{g_1} = 0$. However, after applying reward shaping, the reward for the transition $s_i \rightarrow_{a_1} g_1$ is $r_{g_1} = 1000$, which is higher than the one from $s_i \rightarrow_{a_1} g_2$, which is $r_{g_2} = 110$. This time, the optimal policy should prefer the transition towards g_1, although the *true* optimal policy (i.e. with no reward shaping) should prefer the transition towards g_2.

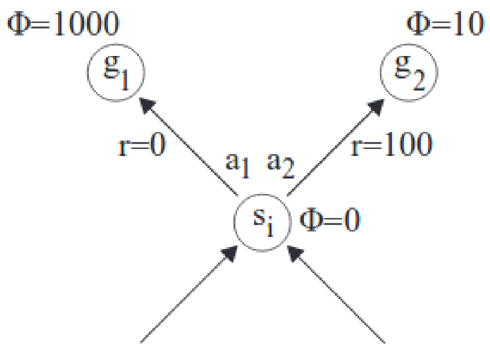


Figure 6.1. An example that shows why the potential function over terminal state must be 0.

More formally, consider the return of the sequence $\bar{s}$, similarly as Equation 4.1:

$$G(\bar{s}) \coloneqq \sum_{k=0}^{N-1} \gamma^k R(s_k, s_{k+1}) \tag{6.7}$$

If we apply PBRS, it becomes:

$$\begin{aligned}
G_\Phi(\bar{s}) &= \sum_{k=0}^{N-1} \gamma^k (R(s_k, s_{k+1}) + F(s_k, s_{k+1})) \\
&= \sum_{k=0}^{N-1} \gamma^k (R(s_k, s_{k+1}) + \gamma\Phi(s_{k+1}) - \Phi(s_k)) \\
&= \sum_{k=0}^{N-1} \gamma^k R(s_k, s_{k+1}) + \sum_{k=1}^{N} \gamma^k \Phi(s_k) - \sum_{k=0}^{N-1} \gamma^k \Phi(s_k) \\
&= G(\bar{s}) + \sum_{k=1}^{N-1} \gamma^k \Phi(s_k) + \gamma^N \Phi(s_N) - \sum_{k=1}^{N-1} \gamma^k \Phi(s_k) - \Phi(s_0) \\
&= G(\bar{s}) + \gamma^N \Phi(s_N) - \Phi(s_0)
\end{aligned} \tag{6.8}$$

The term $\Phi(s_0)$ cannot alter the policy since does not depend from any action executed; on the other hand, the term $\gamma^N \Phi(s_N)$ depends on actions, since the terminal state depends from the previous transitions, hence this term can modify the optimal policy. This happens whenever $\Phi(s_N) \neq 0$, where s_N is any state in which an episode ends, namely goal states, failure states and states where the episode ends due to the time limit exceeded.

A simple solution to this problem is to require that $\Phi(s_N) = 0$ whenever the reinforcement learning trajectory is terminated at state s_N. Notice that if in other trajectories the same s_N is visited, $\Phi(s_N)$ might be different from 0. The only requirement is that if s is a terminal state, then $\Phi(s) = 0$.

Example 6.1. Recalling Example 4.1, we can apply PBRS by defining a potential function that measures how far the agent is from the goal. As heuristic, we can use the *Manhattan distance* between the current position and the goal state. More formally, considering s_{34} the goal state and s_{ij} the current state, we define:

$$\Phi(s) = -[(3 - i) + (4 - j)]$$

It is easy to see that the nearer the agent to the goal state, the higher the potential function evaluated in the state of the agent. For instance, if the current state is s_{11}, $\Phi(s_{11}) = -5$, whereas in s_{33} (which is nearer to the goal), $\Phi(s_{33}) = -1$. With this definition, transition from s_{11} to s_{12} (which makes the agent closer to the goal) evaluates $F(s_{11}, s_{12}) = \Phi(s_{12}) - \Phi(s_{11}) = -4 - (-5) = +1$, whereas for transition $s_{33} \rightarrow s_{32}$ (which makes the agent more distant to the goal), $F(s_{33}, s_{32}) = (-2) - (-1) = -1$.

In the following sections, we will take into account the topics just described in designing a reward shaping strategy for our setting.

6.2 Off-line Reward shaping over $\mathcal{A}_\varphi$

In this part we propose an automatic way to apply reward shaping in the setting presented in Section 5.3. Recall that the state space of $\mathcal{M}_{ag}^{new}$ is $Q_1 \times \ldots Q_m \times S$, where Q_i is the set of states of $\mathcal{A}_{\varphi_i}$, the automaton associated to the LTL$_f$/LDL$_f$ formula φ_i (see Section 2.5).

The basic intuition about our approach is that *every step toward the satisfaction of a goal formula φ_i should be rewarded*, analogously as it is done in reward shaping for classical goals (see Example 6.1). Hence, for a given temporal specification φ_i we should assign to every $q \in Q_i$ a potential function that is, to some extent, inversely proportional to the distance from any final state.

Given a (minimal) automaton $\mathcal{A}_\varphi$ from a LTL$_f$/LDL$_f$ formula φ and its associated reward r, Algorithm 6.1 shows how the potential function is build from $\mathcal{A}_\varphi$. This operation is made *off-line*, i.e. before the learning process. Then we associate automatically to the states of the DFA a potential function $\Phi(q)$ whose value decreases proportionally with the minimum distance between the automaton state q and any accepting state. By construction, potential-based reward shaping with this definition of the potential function gives a positive reward when the agent performs an action leading to a q' that is one step closer to an accepting state, and a negative one in the opposite case.

Notice that, by construction, $G(\bar{s}) = G_\Phi(\bar{s})$, where G_Φ is defined in Equation 6.8. Indeed, $\gamma^N \Phi(s_N) = 0$ because we take into account the issue explained in Section 6.1.3, and $\Phi(s_0) = 0$ by construction of the Algorithm 6.1. Observe that $G(\bar{s}) = G_\Phi(\bar{s})$ is not necessary to guarantee. It is a measure to make our reward shaping the less pervasive as possible for the learnt Q function.

Algorithm 6.1. Off-line Reward Shaping over $\mathcal{A}_\varphi$

```
1: input: (minimal) automaton $\mathcal{A}_\varphi$, reward $r$
2: output: potential function $\Phi : Q \to \mathbb{R}$
3: Let sink be the sink state after the completion of $\mathcal{A}_\varphi$
4: Let $n_{q_0}$ be minimum number of hops to reach an accepting state from $q_0$
5: for $q \in Q$ do:
6:     Let $n_q$ be minimum number of hops to reach an accepting state from $q$
7:     if $n_{q_0} \neq 0$ then                    ▷ i.e. if $q_0$ is NOT an accepting state
8:         $\Phi(q) \leftarrow \frac{n_{q_0} - n_q}{n_{q_0}} \cdot r$
9:     else
10:        $\Phi(q) \leftarrow (n_{q_0} - n_q) \cdot r$
11:    end if
12: end for
13: Let $n_{max}$ the maximum number of hops to reach an accepting state $+1$
14: $\Phi(sink) \leftarrow \frac{n_{q_0} - n_{max}}{n_{q_0}} \cdot r$ if $n_{q_0} \neq 0$ else $(n_{q_0} - n_{max}) \cdot r$
15: return $\Phi$
```

In the actual implementation of the Algorithm 6.1, $\Phi(q)$ can be computed by a least-fix point over the automaton $\mathcal{A}_\varphi$, i.e. starting from the accepting states and then explore the states from the nearest to the farthest ones.

Example 6.2. Consider the following LTL$_f$ goal:

$$\Diamond((A \wedge \neg B) \wedge \Diamond(\neg A \wedge B)) \tag{6.9}$$

and consider its associated automaton, shown in Figure 6.2.

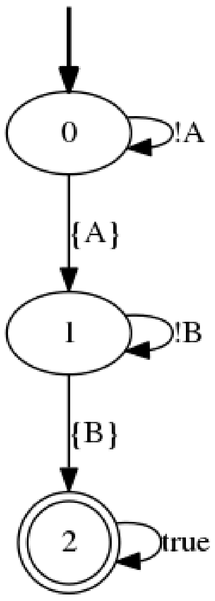


Figure 6.2. The automaton associated to Formula 6.10.

The potential function computed for this automaton, with $r = 200$, would be:

$$\begin{aligned}\Phi(0) =& \frac{2-2}{2} \cdot 200 = & 0\\ \Phi(1) =& \frac{2-1}{2} \cdot 200 = & 100\\ \Phi(2) =& \frac{2-0}{2} \cdot 200 = & 200\end{aligned}$$

Now consider the following transitions:

- $0 \to 0$ gives a shaping reward $F(0, 0) = \Phi(0) - \Phi(0) = 0$
- $0 \to 1$ gives a shaping reward $F(0, 1) = \Phi(1) - \Phi(0) = 100$
- $1 \to 1$ gives a shaping reward $F(1, 1) = \Phi(1) - \Phi(1) = 0$
- $1 \to 2$ gives a shaping reward $F(0, 1) = \Phi(2) - \Phi(1) = 100$

The computations are made by considering that $n_{q_0} = 2$ (i.e. at minimum two transitions that lead to an accepting state, namely $0 \to 1$ and $1 \to 2$) and that $n_1 = 1$. For state 2, $n_2 = 0$ because it is an accepting state.

Example 6.3. Consider the following LDL$_f$ goal:

$$\begin{aligned}&\langle (A \wedge \neg B \wedge \neg C \wedge \neg D)^*;\\&(\neg A \wedge \quad B \wedge \neg C \wedge \neg D)^*;\\&(\neg A \wedge \neg B \wedge \quad C \wedge \neg D)^*;\\&(\neg A \wedge \neg B \wedge \neg C \wedge \quad D)\rangle end\end{aligned} \tag{6.10}$$

and consider its associated (complete) automaton, shown in Figure 6.3.
The potential function computed for this automaton, with $r = 100$, would be:

$$\begin{aligned}\Phi(0) &= \frac{1-1}{1} \cdot 100 = \quad 0\\ \Phi(3) &= \frac{1-1}{1} \cdot 100 = \quad 0\\ \Phi(2) &= \frac{1-1}{1} \cdot 100 = \quad 0\\ \Phi(1) &= \frac{1-0}{1} \cdot 100 = \quad 100\\ \Phi(sink) &= \frac{1-2}{1} \cdot 100 = -100\end{aligned}$$

Now consider every possible transition from 0:

- $0 \rightarrow 0$ gives a shaping reward $F(0, 0) = \Phi(0) - \Phi(0) = 0$
- $0 \rightarrow 3$ gives a shaping reward $F(0, 3) = \Phi(3) - \Phi(0) = 0$
- $0 \rightarrow 2$ gives a shaping reward $F(0, 2) = \Phi(2) - \Phi(0) = 0$
- $0 \rightarrow 1$ gives a shaping reward $F(0, 1) = \Phi(1) - \Phi(0) = 100$
- $0 \rightarrow sink$ gives a shaping reward $F(0, sink) = \Phi(sink) - \Phi(0) = -100$

The computations are made by considering that $n_{q_0} = 1$ due to transition $0 \rightarrow_{\{D\}} 3$ and that $n_q = 1$ also fo states 2 and 3. For state 1, $n_1 = 0$ because it is an accepting state. Notice that, since states $0, 3, 2$ are at the same distance to the (unique) accepting state 1, they have the same potential, i.e. 0. Thus, only transitions to the accepting state 1 are rewarded. Moreover, notice that $n_{sink} = n_{max} + 1 = n_{q_0} + 1 = 2$. Hence, by construction, *sink* is the state at the lowest potential.

6.3 On-The-Fly Reward shaping

Reward shaping can also be used when the DFAs of the LTL$_f$/LDL$_f$ formulas are constructed *on-the-fly* (Brafman et al., 2018) so as to avoid to compute the entire automaton off-line. To do so we can rely on dynamic reward shaping (see Section 6.1.2). The idea is to build $\mathcal{A}_\varphi$ progressively while learning. During the learning process, at every step, the value of the fluents $\ell \in \mathcal{L}$ is observed and the successor state q' of the current state q of the DFA on-the-fly is computed. Then, the transition and the new state just observed are added into the built automaton at time t, $\mathcal{A}_{\varphi,t}$, yielding $\mathcal{A}_{\varphi,t'}$. The potential function Φ

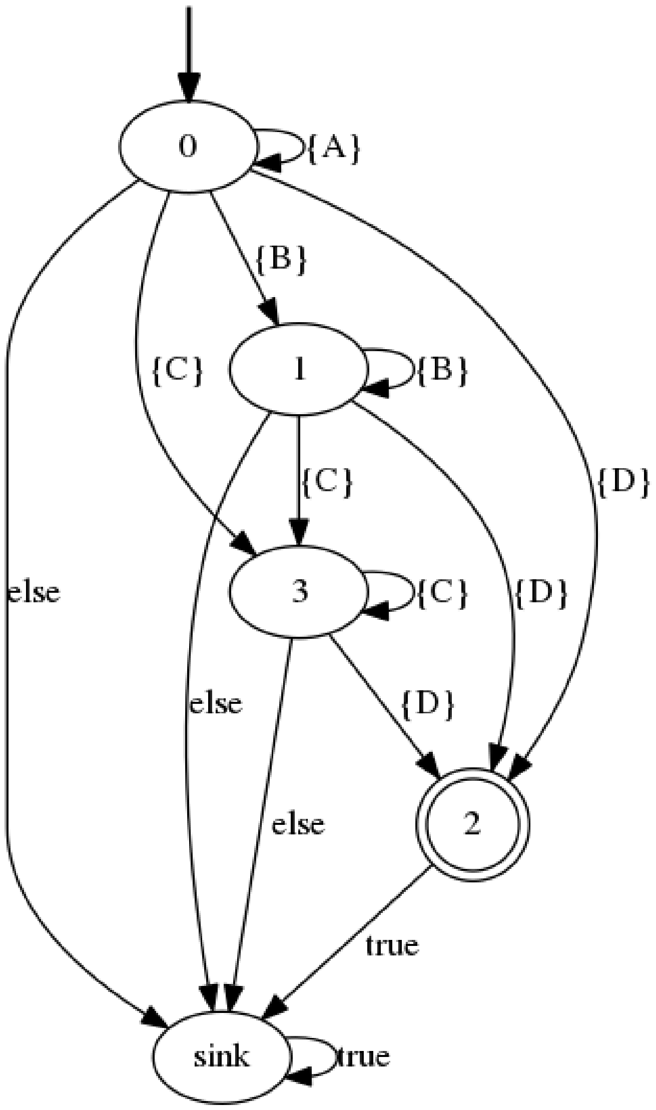


Figure 6.3. The automaton associated to Formula 6.10.

for $\mathcal{A}_{\varphi,t'}$ is recomputed for the new version of the automaton. In this case, the shaping function takes the following form:

$$F(q,t,a,q',t') = \Phi(q',t') - \Phi(q,t) \tag{6.11}$$

i.e. the dynamic reward shaping in Equation 6.4 but taking into account the issue presented in Section 6.1.3 where $\Phi(q,t)$ is a variant of the off-line case, but computed on the automaton $\mathcal{A}_{\varphi,t}$. In the following we explain how Φ in the on-the-fly case differs from the one shown in Section 6.2.

Details about $\Phi(q, t)$

There is an important issue which has not yet been pointed out. In the *off-line* variant described in Section 6.2, we apply reward shaping at every transition by knowing the full automaton $\mathcal{A}_\varphi$; however, in the *on-the-fly variant*, at the beginning of the learning task, we have an "empty" automaton, i.e. only the initial state with no transition from it. Let assume that after an action the automaton makes a move from the initial state. How can we assign a positive/negative shaping reward on the transition if we do not know the goodness of the transition? And if during a simulation we discover an accepting state, there could be a nearer accepting state that has not yet been discovered, but in order to reach it, we should first discover other intermediate states. How to allow the agent to discover it while not fixing on the only known accepting states?

It is clear that the farther the learning task goes, the more accurate will be the shaping rewards because every observed transition of the automaton during the activity of the agent is stored and eventually the entire automaton will be explored. However, some paths ending in an accepting state are not fully explored, so how to encourage the agent, in our setting, to explore those paths?

In order to determine $\Phi(s, t)$ for a given $A_{\varphi,t}$, we consider the same computations of Algorithm 6.1 but *considering the leaves (wrt the initial state) of the automaton as accepting state*. In this way, even if a path does not end in an accepting state, its following is still rewarded. It could be *wrongly rewarded*, since the path might lead to a failure state. However, the dynamic reward shaping theory states that until the potential-based condition is preserved, also the optimal and near-optimal solutions are preserved; moreover, eventually, once the final state of the path is discovered, the reward for that path will assume the right values.

In this setting, a *leaf state* is a state that is at the "boundaries" of the automaton; that is, a state that requires a high number of transitions and that their outer edges does not end in a farthest state from the initial state. The search for the leaf states is done through Breadth-First Search, while the computation is the same as the Algorithm 6.1 by considering as accepting state the *true* accepting states, discovered until now, and the leaf states computed as just explained.

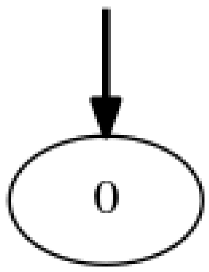
Example 6.4. Let consider Example 6.2, but with the on-the-fly construction of the potential function just described. Consider the following trace:

$$\pi = \langle \{\}, \{A\}, \{A\}, \{B\}, \{A\} \rangle$$

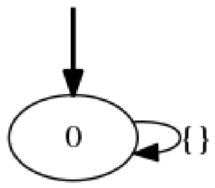
The updates corresponding to each trace symbol are shown in Figure 6.4. Notice that at each relevant change of the automaton's topology, also the potential function changes. Moreover, consider the step in Figure 6.4c. In order to compute the potential function, it has been used the Algorithm 6.1 by considering the state 1 to be an accepting one. This implies an overestimation of the true potential function value (i.e. 100), but it is useful to encourage the agent to explore that path of the automaton. Once a further state is discovered, state 1 is not considered anymore as fake accepting state: indeed, from step in Figure 6.4e, its value is the true one, i.e. $\Phi(1) = 100$.

It is easy to see that:

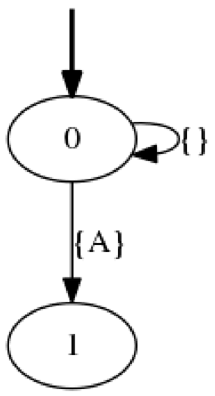
Theorem 6.2. *Automata-based reward shaping, both in off-line and on-the-fly variants, preserves optimality and near-optimality of the MDP solutions.*



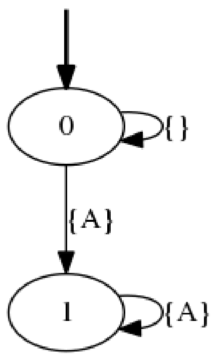
(a) $\Phi(0) = 0$



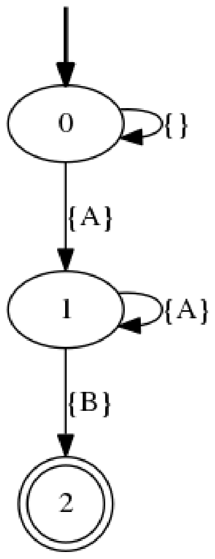
(b) After $\{\}$: $\Phi(0) = 0$



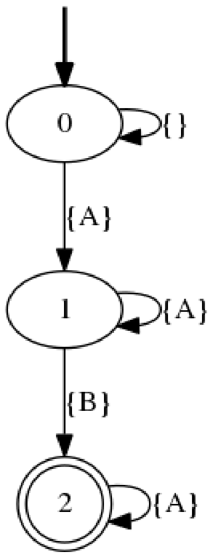
(c) After $\{A\}$: $\Phi(1) = 200$



(d) After $\{A\}$: nothing changes.



(e) After $\{B\}$: $\Phi(1) = 100$ $\Phi(2) = 200$



(f) After $\{A\}$: nothing changes.

Figure 6.4. An example of automaton and potential function built from scratch. The formula is the one used for Example 6.2.

Proof. For the off-line case, the shaping-reward function Φ is, by construction, potential based, hence fulfilling the premises of theorems in (Ng et al., 1999) and (Grześ, 2017). Also for the on-the-fly variant, we observe that our construction is compliant with the requirements defined in (Devlin and Kudenko, 2012). □

6.4 Conclusion

In this chapter, we propose an approach to apply reward shaping in the setting described in Chapter 4. We first revised the background of reward shaping theory. Then we describe an automatized way to define Φ, by leveraging the structure of the automaton $\mathcal{A}_\varphi$ associated to the LTL$_f$/LDL$_f$ goal φ. We devised a variant in which the automaton is not known apriori, but it is built from scratch during learning, and the potential function is dynamically updated. Finally, we observe that both the designed reward shaping preserve policy invariance.

Chapter 7

RLTG

In this chapter, we describe RLTG (Reinforcement Learning for Temporal Goals), a software project written in Python. It is the reference implementation of many of the topics described in Chapter 4 and Chapter 6.

7.1 Introduction

Main features: RLTG is a Python framework that allows you to:

- Setup a classic reinforcement learning task by using OpenAI Gym environments.
- Modularization of a reinforcement learning system: definition of `Environment`, `Agent`, `Brain`, `Policy`, allowing for easy extension;
- Support for Sarsa(λ) and Q(λ) with ϵ-greedy policy.
- Support for LTL$_f$/LDL$_f$ goal specifications, as described in Chapter 4 and Chapter 5.

Dependencies: RLTG requires Python>=3.5 and depends on the following packages:

- FLLOAT, described in Chapter 3;
- Gym OpenAI, a toolkit for developing and comparing reinforcement learning algorithms. It offers a useful abstraction of reinforcement learning environments.

Installation: You can find the package on PyPI, hence you can install it with:

```
pip install rltg
```

The software is open source and is released under MIT license.

7.2 Package structure

The package is structured as follows:

- `agents/`: contains multiple packages and modules to build the learning agent.

- brains/: it contains Brain.py, the basic abstraction of the reinforcement learning algorithm, as well as TDBrain.py, the implementation of Temporal Difference learning (see Section 4.3);
- parameters/ and policies/ contain implementations for eligibility traces and ϵ-greedy policy;
- feature_extraction.py: it contains classes for feature extraction like FeatureExtractor. They leverage the Space abstraction provided by OpenAI Gym. Here you can find many types of state spaces supported.
- Agent.py, the module that contains the Agent abstraction, defining the abstract methods for interact with the environment (e.g. observe for observing the new environment state, step for choose an action). It needs a FeatureExtractor and a Brain.
- TemporalEvaluator.py, which implements the temporal goal specified by an LTL$_f$/LDL$_f$ formula.

- logic/: contains the implementations of reward shaping as described in Section 6.2 and 6.3. The main modules are CompleteRewardAutomaton.py, for the implementation of Off-line Reward Shaping, and PartialRewardAutomaton.py, for the On-the-fly Reward Shaping.
- trainers/: contains the implementation of a highly-customizable training loop, e.g. you can specify under which conditions the training should stop, you can visually render the agent during the learning and you can pause/resume the learning. The GenericTrainer is used for classic reinforcement learning, whereas TGTrainer is used for temporal goal reinforcement learning.

In the next sections we will go in deep with the features provided by RLTG and we will see some code examples.

7.3 Classic Reinforcement Learning with RLTG

RLTG allows you to set up a classic reinforcement learning system. That is, it provides abstractions for the main components of an RL system and also for the training loop.

7.3.1 Environment

The software assumes that the environment object has the OpenAI Gym interface. That is, the environment implements some methods e.g.:

- step(action): observation, reward, done, info. From an action, return the new observation, the reward of the last transition, if the episode is ended and some additional informations. It is implicit that the environment object keeps track of the state of the environment in order to compute the next one.
- reset(): reset the environment at the initial state.

7.3.2 Agent

In Listing 7.1 it is reported the code in `agents/Agent.py` that define the abstract class `Agent`.

Listing 7.1. The `Agent` abstraction.

```python
class Agent(ABC):
  def __init__(self, sensors: RobotFeatureExtractor,
    brain:Brain,
    reward_shapers:Tuple[RewardShaping]=(),
    eval:bool=False):
    """..."""
    if brain.observation_space and \
     sensors.output_space != brain.observation_space:
       raise ValueError("space␣dimensions␣are␣not␣compatible.")
    self.sensors = sensors
    self.brain = brain
    self.reward_shapers = reward_shapers
    self.eval = eval

  def set_eval(self, eval:bool):
    """Setter method for "eval" field."""
    self.eval = eval
    self.brain.set_eval(eval)

  def observe(self, state, action, world_reward, state2):
    """Called at each observation. """
    features_1 = self.sensors(state)
    features_2 = self.sensors(state2)
    reward = world_reward +
    sum(rs(features_1, features_2) for rs in self.reward_shapers)
    return self._observe(features_1, action, reward, features_2)

  def _observe(self, features, action, reward, features2):
    obs = AgentObservation(features, action, reward, features2)
    self.brain.observe(obs)
    return obs

  def update(self):
    """Called at the end of each iteration.
    It MUST be called only once for iteration."""
    self.brain.update()

  def start(self, state):
    features = self.sensors(state)
    return self.brain.start(features)

  def step(self, obs: AgentObservation):
    return self.brain.step(obs)

  def end(self, obs: AgentObservation):
    """Called at the end of each episode.
    It MUST be called only once for episode."""
    self.brain.end(obs)

  def save(self, filepath):
    with open(filepath + "/agent.pkl", "wb") as fout:
      pickle.dump(self, fout)

  @staticmethod
  def load(filepath):
    with open(filepath + "/agent.pkl", "rb") as fin:
      agent = pickle.load(fin)
      return agent

  def reset(self):
    self.brain.reset()
```

As you can see, it is composed by:

- a `RobotFeatureExtractor`, that is used in the `observe` method to extract the relevant features of the world and to generate a new agent's state;
- a `Brain` which implements the reinforcement learning algorithm used by the agent;
- a collection of `RewardShaping` object that implements the reward shaping technique.

The main methods are:

- `set_eval` switches from "training mode" to "evalutation mode". In the former, the brain actually learns from experience and modify its estimation of the action-value function $Q(s, a)$. In the latter, the agent does not modify itself, but rather take the optimal decision (according to what it learnt).
- `observe` take in input a SARS tuple and and returns the observation made by its `Brain`;
- `update`, `start`, `step` and `end` call the analogous methods of the `Brain`. The method `step` actually does the learning step, and returns the next action.
- `save` and `load` are the methods for, respectively, save and load the state of the agent on disk.

7.3.3 `FeatureExtractor`

The abstract class `FeatureExtractor` is defined in `agents/feature_extraction.py`. Listing 7.2 shows the implementation.

Listing 7.2. The `FeatureExtractor` abstraction.

```python
class FeatureExtractor(ABC):

  def __init__(self, input_space: Space, output_space: Space):
    self.input_space = input_space
    self.output_space = output_space

  def __call__(self, input, **kwargs):
    """Extract features from the input and make sanity check
    of the input and the output dimensions of the abstract method
    '_extract'"""

    if not self.input_space.contains(input):
      raise ValueError("input␣space␣dimensions␣are␣not␣correct.")

    output = self._extract(input, **kwargs)

    if not self.output_space.contains(output):
      raise ValueError("output␣space␣dimensions␣are␣not␣correct.")

    return output

  @abstractmethod
  def _extract(self, input, **kwargs):
    """
    Extract the feature from `input`
    (contained into `self.input_space`).
    The features must be contained into `self.output_space`.
    :param  input: a state belonging to input_space
    :returns output: a state belonging to output_space
    """
      raise NotImplementedError
```

A `FeatureExtractor` object (and its extensions) requires both an `input_space` and an `output_space`, in order to do some checks about the inputs and the outputs of the operator. The `_extract` method is abstract and must be implemented by the subclasses. The method `__call__` (which makes the object callable) simply checks if the provided `input` is contained in the input state space; then it calls the `_extract` methods which returns the result; finally, It is checked whether the result is contained in the predefined output space.

7.3.4 Brain

One of the main important abstraction is `Brain`, which tries to generalize how an RL algorithm can work at high level. In Listing 7.3 it is shown the code that implements it, in `agents/brains`.

Listing 7.3. The `Brain` abstraction.

```python
class Brain(ABC):
```

```
"""The class which implements the core of the algorithms"""

def __init__(self, observation_space:Space, action_space:Space,
  policy:Policy):
  """
  :param observation_space: instance of Space or None.
               If None, it means that the observation space
               is not known a priori and so it is not needed
               for the algorithm.
  :param action_space:  instance of Space.
  :param policy:        behavior policy.
  """

  self.observation_space = observation_space
  self.action_space = action_space
  self.policy = policy

  self.episode = 0
  self.iteration = 0
  self.episode_iteration = 0
  self.obs_history = []
  self.total_reward = 0

  self.eval = False

def set_eval(self, eval:bool):
  self.eval = eval
    self.policy.set_eval(eval)

@abstractmethod
def choose_action(self, state, **kwargs):
  """From a state, return the action for the implemented approach.
  e.g. in Q-Learning, select the argmax of the Q-values
   relative to the 'state' parameter."""
    raise NotImplementedError

@abstractmethod
def observe(self, obs:AgentObservation, *args, **kwargs):
    """Called at each observation.
    E.g. in args there can be the S,A,R,S' tuple
    for save it in a buffer
    that will be read in the "learn" method."""
    self.obs_history.append(obs)
    self.total_reward += obs.reward

def update(self, *args, **kwargs):
  """action performed at the end of each iteration
  Subclass implementations should call this method.
```

```
    """
    self.episode_iteration += 1
    self.iteration += 1
    self.policy.update()

  def start(self, state):
    if not self.eval:
      self.episode += 1
    self.episode_iteration = 0
    self.total_reward = 0
    self.policy.reset()
    self.obs_history = []

  @abstractmethod
  def step(self, obs: AgentObservation, *args, **kwargs):
    """The method performing the learning
    (e.g. in Q-Learning, update the table)"""
    raise NotImplementedError

  def end(self, obs:AgentObservation, *args, **kwargs):
    """action performed at the end of each episode
    Subclasses implementations should call this method.
    """
    self.episode_iteration += 1

  @abstractmethod
  def reset(self):
      raise NotImplementedError
```

You can notice that the abstract methods do some utility computations (e.g. track of the numbers of episodes, number of steps per episode, total reward collected in the last episode etc.) but some of them has to be expanded in order to implement the core functionality of a given RL algorithm. It is the case of `choose_action`, which returns optimal action, `observe` which is called from the class `Agent` at each observation (after the feature extraction), `step` which actually implements the learning.

A `Brain` requires an `observation_space` and an `action_space` that define the dimensions of the task, and `policy` which is used for determine the next action to take (e.g. in the `EGreedy` policy, it chooses whether the action must be taken optimally or randomly).

7.3.5 Trainer

The `Trainer` class implements the RL training loop.

Listing 7.4. The `Trainer` that implements the training loop.

```
class GenericTrainer(Trainer):

  def __init__(self, env:GoalEnvWrapper, agent:Agent, n_episodes=1000,
        data_dir=DEFAULT_DATA_DIR,
```

```python
        stop_conditions=(GoalPercentage(10, 1.0), ),):
    self.env = env
    self.n_episodes = n_episodes
    self.stop_conditions = list(stop_conditions)
    self.agent = agent

    self.stats = StatsManager(name="train_stats")
    self.optimal_stats = StatsManager(name="eval_stats")


  def main(self, eval:bool=False, render:bool=False, verbosity:int=1):

    ...

    if eval:
      stats.reset()
      optimal_stats.reset()
      self.cur_episode = 0

    for ep in range(self.cur_episode, num_episodes):

      if not eval:
        steps, total_reward, goal = self.train_loop(render=render)
        stats.update(steps, len(agent.brain.Q), total_reward, goal)
        ...

      # try optimal run
      agent.set_eval(True)
      steps, total_reward, goal = self.train_loop(render=render)
      optimal_stats.update(steps, len(agent.brain.Q),
        total_reward, goal)
      ...
      agent.set_eval(False)


      if self.check_stop_conditions(optimal_stats, eval=eval):
        break

      if self.cur_episode%100==0 and not eval:
        agent.save(self.agent_data_dir)
        self.save()

      self.cur_episode = ep

    if not eval:
      self.save()
      agent.save(self.agent_data_dir)
      ...
    return stats, optimal_stats


  def train_loop(self, render:bool=False):
    env = self.env
    agent = self.agent

    stop_condition = False
    info = {"goal": False}

    state = env.reset()
    action = agent.start(state)
    obs = None

    while not stop_condition:
      state2, reward, done, info = env.step(action)
      if render: env.render()

      stop_condition = self.check_episode_stop_conditions(done,
        info.get("goal", False))
      obs = agent.observe(state, action, reward, state2,
        is_terminal_state=stop_condition)

      if stop_condition:
        break

      action = agent.step(obs)
      agent.update()
      state = state2

    agent.end(obs)
    steps, tot_reward = agent.brain.episode_iteration,
      agent.brain.total_reward
    return steps, tot_reward, self.is_goal(info)

  def check_stop_conditions(self, stats:StatsManager, eval=False):
    return not eval and \
      all([s.check_condition(stats_manager=stats)
        for s in self.stop_conditions])

  def check_episode_stop_conditions(self, done, goal):
    return done or goal

  def is_goal(self, info, *args, **kwargs):
    return info.get("goal", False)

  def save(self):
    with open(self.data_dir + "/trainer.pkl", "wb") as fout:
      pickle.dump(self, fout)

  def reset(self):
    self.cur_episode = 0
    self.stats.reset()
    self.optimal_stats.reset()
```

The main method `main` starts and manages the learning process. It can be run in training mode (`eval=False`), where the agent actually learns, and in evaluation mode (`eval=True`), where there is no learning at all, but the agent just chooses the optimal actions. There is the main for-loop that iterates for a number of episode specified by the user.

At each iteration it is executed an episode by calling the method `train_loop`. This method does the following steps:

- Reset the environment and the agent.
- While the stop condition is not true: take an action, observe the new state and from the SARS tuple make an agent step (i.e. learn from the observation).
- When the stop condition is true, the episode is terminated, and the control is passed to the `main` method.

The stop conditions are specified when constructing the `GeneralTrainer` object. They uses the statistics collected during the simulations and could be to reach a threshold of average reward, or of the percentage of goals reached.

In the main loop, for each episode, two simulations are executed: one where the behavior policy is followed and where the agent learns, and the other when, during the simulation, the optimal policy is adopted. The evaluation of the stop conditions is done by using the statistics from the optimal run.

7.4 Reinforcement Learning for Temporal Goals

In this section we show how to use RLTG for setting up a RL system as described in previous chapters (i.e. Chapter 4, 5 and 6).

7.4.1 TGTrainer

Analogously to `GeneralTrainer` described before, we introduce the `TGTrainer` (temporal goal trainer), which allow to run a *RL for* LTL$_f$/LDL$_f$ *goals* task. Its implementation is shown in Listing 7.5.

Listing 7.5. `TGTrainer`

```
class TGTrainer(GenericTrainer):
  """Temporal Goal Trainer"""

  def __init__(self, env:GoalEnvWrapper, agent:TGAgent=None,
      n_episodes=1000,
      data_dir="data",
     stop_conditions=(GoalPercentage(10, 1.0), ),
  ):
  super().__init__(env, agent, n_episodes, data_dir, stop_conditions)
  self.stop_conditions.append(CheckAutomataInFinalState())

  def train_loop(self, render:bool=False):
  env = self.env
  agent = self.agent

  stop_condition = False
  info = {"goal": False}

  state = env.reset()
  action = agent.start(state)
  obs = None
  if render: env.render()

  while not stop_condition:
    state2, reward, done, info = env.step(action)
    if render: env.render()

    old_automata_state = agent.get_automata_state()
    new_automata_state = agent.sync(action, state2)

    stop_condition = self.check_episode_stop_conditions(done,
     info.get("goal", False))
    obs = agent.observe(state, action, reward, state2,
        old_automata_state=old_automata_state,
        new_automata_state=new_automata_state,
        is_terminal_state=stop_condition)

    if stop_condition:
    break

    action = agent.step(obs)
    agent.update()
    state = state2

  agent.end(obs)
  steps, tot_reward = agent.brain.episode_iteration,
  agent.brain.total_reward
  return steps, tot_reward, self.is_goal(info)


  def check_stop_conditions(self, stats:StatsManager, eval=False):
  return not eval and\
    all([s.check_condition(
      stats_manager=stats,
      temporal_evaluators=self.agent.temporal_evaluators
      ) for s in self.stop_conditions])

  def check_episode_stop_conditions(self, done, goal):
  temp_evals = self.agent.temporal_evaluators
  any_te_failed = any(t.is_failed() for t in temp_evals)
  all_te_true = all(t.is_true() for t in temp_evals)\
    if len(temp_evals) > 0 else False

  if any_te_failed:
    logging.debug("some␣automaton␣in␣failure␣state:␣end␣episode")

  return done or any_te_failed or (goal and all_te_true)

  def is_goal(self, info, *args, **kwargs):
  return super().is_goal(info) and\
    all(t.is_true() for t in self.agent.temporal_evaluators)
```

`TGTrainer` extends `GenericTrainer` and reimplements some methods from the parent class:

- `train_loop` is adapted so that are included the automata states and the synchronization of the automata.
- `check_stop_conditions` of the main loop includes also the temporal evaluators objects;
- `check_episode_stop_conditions` is defined as explained in Section 5.4.
- `is_goal` is adapted so that we check if every automaton $\mathcal{A}_{\varphi_i}$ is in an accepting state.

7.4.2 TGAgent

The `TGAgent` class extends the abstract `Agent` class, and implements an agent that learns temporal goals. In Listing 7.6 you can see the code.

Listing 7.6. `TGAgent`

```
class TGAgent(Agent):
  """Temporal Goal agent"""

  def __init__(self,
        sensors: RobotFeatureExtractor,
        brain: Brain,
        temporal_evaluators: List[TemporalEvaluator],
        reward_shaping: bool = True):
    assert len(temporal_evaluators) >= 1

    ...

```

```
def sync(self, action, state2):
  new_automata_state =
    tuple(temp_eval.update(action, state2)
      for temp_eval in self.temporal_evaluators)
  return new_automata_state

def start(self, state):
  for t in self.temporal_evaluators:
    t.reset()
  automata_states = [temp_eval.get_state()
    for temp_eval in self.temporal_evaluators]
  transformed_state = self.state_extractor(state, automata_states)
  # return super().start(transformed_state)
  return self.brain.start(transformed_state)

def state_extractor(self, world_state, automata_states: List):
  # the state is a tuple: (features, A_1 state, ..., A_n state)
  # A_i is the i_th automaton associated to the i_th temporal goal
  state = tuple([self.sensors(world_state)] + list(automata_states))
  return state

def reward_extractor(self, world_reward, automata_rewards: List):
  res = world_reward + sum(automata_rewards)
  return res

def get_automata_state(self):
  return tuple(temp_eval.get_state()
    for temp_eval in self.temporal_evaluators)

def observe(self, state, action, reward, state2,
      old_automata_state=(), new_automata_state=(),
      is_terminal_state=False):

  automata_rewards = [te.reward \
    if is_terminal_state and te.is_true() else 0
    for te in self.temporal_evaluators]

  # apply reward shaping to the observed automata rewards
  for i in range(len(self.reward_shapers)):
    plus = self.reward_shapers[i](old_automata_state[i],
                new_automata_state[i],
                is_terminal_state=is_terminal_state)
    automata_rewards[i] += plus

  old_state = self.state_extractor(state, old_automata_state)
  new_state2 = self.state_extractor(state2, new_automata_state)
  new_reward = self.reward_extractor(reward, automata_rewards)

```

```
    ...

    return super()._observe(old_state, action, new_reward, new_state2)

  def save(self, filepath):
    super().save(filepath)
    for idx, te in enumerate(self.temporal_evaluators):
      with open(filepath + "/te%02d.dump" % idx, "wb") as fout:
        te.reset()
        pickle.dump(te, fout)

  @staticmethod
  def load(filepath):
    ...

  def set_eval(self, eval: bool):
    super().set_eval(eval)
    for te in self.temporal_evaluators:
      te.set_eval(eval)

  def reset(self):
    super().reset()
    for te in self.temporal_evaluators:
      te.reset()
```

The `TGAgent` object requires the sensors and a brain as the class `Agent`. Moreover, it includes a list of `TemporalEvaluator`, which manages the temporal goal associated to a formula. Notice that:

- Some of the methods of the parent class are wrapped or reimplemented in order to be adapted in the new settings. Some examples are: `observe` that now includes the cartesian product between low-level states and automata states.
- The `sync` method is used to update every automata after a transition.

7.4.3 TemporalEvaluator

The `TemporalEvaluator` object implements what is needed to keep track the $\text{LTL}_f/\text{LDL}_f$ goal and the associated automaton, as well as the automata-based reward shaping.

Listing 7.7. `TemporalEvaluator`

```
class TemporalEvaluator(ABC):
  def __init__(self, goal_feature_extractor:FeatureExtractor,
               alphabet:Set[Symbol],
               formula:LDLfFormula, reward,
               gamma=0.99, on_the_fly=False):
    self.goal_feature_extractor = goal_feature_extractor
    self.alphabet = Alphabet(alphabet)
    self.formula = formula
    self.reward = reward
    self.on_the_fly = on_the_fly
    if not on_the_fly:
      self._automaton = CompleteRewardAutomaton._fromFormula(alphabet,
              formula, reward, gamma=gamma)
      self.simulator = RewardAutomatonSimulator(self._automaton)
    else:
      self.simulator = PartialRewardAutomaton(self.alphabet,
              self.formula, reward, gamma=gamma)

    self.eval = False

  @abstractmethod
  def fromFeaturesToPropositional(self, features, action) -> Set[Symbol]:
    raise NotImplementedError

  def update(self, action, state):
    """update the automaton.
    :param action: the action to reach the state
    :param state: the new state of the MDP
    :returns (new_automaton_state, reward)"""
    features = self.goal_feature_extractor(state)
    propositional = self.fromFeaturesToPropositional(features, action)
    old_state = self.simulator.get_current_state()
    new_state = self.simulator.make_transition(propositional,
        eval=self.eval)
    return self.simulator.get_current_state()


  def get_state(self):
    return self.simulator.get_current_state()

  def get_state_space(self):
    if not self.on_the_fly:
      return Discrete(len(self.simulator.state2id))
    else:
      # estimate the state space size.
      # the estimate MUST overestimate the true size.
      # return Discrete(100)
      return None

  def reset(self):
    self.simulator.reset()

  def is_failed(self):
    return self.simulator.is_failed()

  def is_true(self):
    return self.simulator.is_true()

  def is_terminal(self):
    return self.is_true() or self.is_failed()

  def set_eval(self, eval:bool):
    self.eval = eval
```

The Temporal Evaluator is an abstract class composed by:

- A FeatureExtractor which generates a high-level representation of the world state;
- A set of Symbols that are used from the automaton and to specify the formulas.
- The reward associated to the temporal goal of interest.
- The on_the_fly parameter which switch from off-line reward shaping and on-the-fly reward shaping, associated respectively to CompleteRewardAutomaton class and PartialRewardAutomaton class, both defined in the logic/ package.

The implementations of a Temporal Evaluator must implement the method fromFeaturesToProposi which take as input the features of a state and the action taken and returns a set of symbols that determine the transition of the automaton in the update method.

7.5 Code examples

7.5.1 Classic Reinforcement Learning

Here we will see an example of reinforcement learning task using RLTG with the OpenAI Gym environment Taxi-v2 (Dietterich, 1998), available here.

Listing 7.8. Classic Reinforcement Learning using RLTG

```python
import gym

from rltg.agents.RLAgent import RLAgent
from rltg.agents.brains.TDBrain import Sarsa
from rltg.agents.feature_extraction\
 import IdentityFeatureExtractor
from rltg.agents.policies.EGreedy import EGreedy
from rltg.trainers.GenericTrainer import GenericTrainer
from rltg.utils.GoalEnvWrapper import GoalEnvWrapper
from rltg.utils.StoppingCondition\
 import AvgRewardPercentage

def taxi_goal(*args):
  reward =args[1]
  done =args[2]
  return done and reward == 20

if __name__ == '__main__':
```

```
19 env = gym.make("Taxi-v2")
20 env = GoalEnvWrapper(env, taxi_goal)
21
22 observation_space = env.observation_space
23 action_space = env.action_space
24 print(observation_space, action_space)
25 agent = RLAgent(
26 IdentityFeatureExtractor(observation_space),
27 Sarsa(observation_space, action_space,
28     EGreedy(0.1), alpha=0.1, gamma=0.99, lambda_=0.0)
29 )
30
31 tr = GenericTrainer(
32   env,
33   agent,
34   n_episodes=10000
35   stop_conditions=(
36     AvgRewardPercentage(window_size=100, target_mean=9.0)
37   )
38 )
39 tr.main()
```

Now we analyze some of the APIs provided by RLTG:

- In line 19 we defined the environment by OpenAI Gym APIs. In line and 20 we wrapped with a function that make it *goal-based*.
- In line 25 we defined our reinforcement learning agent. We had to specify a `FeatureExtractor`, i.e. which features of the state provided from the environment we consider relevant, and a `Brain`, i.e. the responsible for the learning of the policy. In particular:
  - `IdentityFeatureExtractor` extends `FeatureExtractor` and it simply get the entire state space from the environment, without modifying it; It requires the observation state space of the environment to do some sanity checks.
  - `Sarsa` extends `TDBrain`, and implements Sarsa(λ) (see Section 4.3). It requires:
    - the observation space where the algorithm learns (in this case is the same of the one provided by the environment);
    - the action space from where actions can be selected;
    - the behavior policy (in this case ϵ-greedy policy with $\epsilon = 0.1$.
    - the learning rate $\alpha = 0.1$, the discount factor $\gamma = 0.99$, the eligibility trace parameter $\lambda = 0$.
  - In line 31 we configured the manager of the training task. `GenericTrainer`, which extends from `Trainer`, is configured by the environment `env` and the agent `agent` defined before, the maximum number of episodes `n_episodes` and the stop conditions. `AvgRewardPercentage` with parameters `window_size` $= 100$ and `target_mean` $= 9.0$ means that we will stop the training if in the last 100 episodes an average reward of 9.0 is observed.

If you run the script, you will get some statistics for each episode, e.g. the number of explored states, the total reward observed, if the goal has been reached.

7.5.2 Temporal goal Reinforcement Learning

Now we will see how to use RLTG for a temporal goal specified by a LTL$_f$/LDL$_f$ formula. In particular, we will use the BREAKOUT environment, presented in Example 4.2 and that we will further describe in Chapter 8. In the next, we show an excerpt of this script:

Listing 7.9. Temporal Goal Reinforcement Learning using RLTG

```
1  ...
2  ...
3  ...
4  if __name__ == '__main__':
5    env = GymBreakout(brick_cols=3, brick_rows=3)
6
7    gamma = 0.999
8    on_the_fly = False
9    reward_shaping = False
10
11   agent = TGAgent(
12     BreakoutNRobotFeatureExtractor(env.observation_space),
13     Sarsa(None, env.action_space, policy=EGreedy(0.1),
14       alpha=0.1, gamma=gamma, lambda_=0.99
15     ),
16     [BreakoutCompleteColumnsTemporalEvaluator(
17       env.observation_space, bricks_rows=env.brick_rows,
18       bricks_cols=env.brick_cols, left_right=True,
19       gamma=gamma, on_the_fly=on_the_fly)
20     ],
21     reward_shaping=reward_shaping
22   )
23
24   tr = TGTrainer(env, agent, n_episodes=2000,
25     stop_conditions=(GoalPercentage(100, 0.2),),
26   )
27
28   stats, optimal_stats = tr.main()
```

Observe that:

- In Listing 7.8 we used `RLAgent`, while in this case, in line 11, we use `TGAgent`. The arguments provided in the constructors are the same of the previous example except for an additional argument which is the list of `TemporalEvaluator` (line 16). Notice that this definition of *list of temporal goals* is compliant with Definition 5.1. We do not discuss here details about this particular temporal evaluator, i.e. `BreakoutCompleteColumnsTemporalEvaluator`. Observe that we can specify if the temporal evaluator should use a off-line automaton or a on-the-fly automaton by the boolean parameter `on_the_fly`. Finally, notice the `reward_shaping` flag to toggle automata-based reward shaping (Section 6.2 and 6.3).

- Another difference is in using `TGTrainer` instead of `GenericTrainer`. This version is more specialized to deal with the setting explained in Section 5.3. The signature of the constructor is the same of `GenericTrainer`. Notice that in this particular case we used `GoalPercentage(100, 0.2)`, which means we require the agent have reached the goal at least 20 times in the last 100 episodes.

7.6 Conclusion

In this chapter we described RLTG, a Python package to implement a reinforcement learning system in a modular way, both for classic tasks and for the setting described in previous chapters. We've seen the implementation of the main classes of the software and provided some typical use case with code examples.

Chapter 8

Experiments

In this chapter we give experimental evidence of the goodness of our approach, discussed mainly in Chapter 4 and 6, by using the implementations presented in Chapter 3 and 7. For each considered environment, we give a formal description in the framework of MDPs and assign temporal goals to be satisfied, as explained in earlier chapters.

8.1 BREAKOUT

In this section, we use the BREAKOUT environment (already presented in Example 4.2). We first show how the combination of temporal goals is successfully learnt by the agent. Then, we show a benchmarking between the use of off-line reward shaping, on-the-fly reward shaping and no reward shaping.

Description of the Environment: The actions available to the agent are *left*, *right* and *no-action*. The relevant features are: position of the paddle p_x, position of the ball b_x, b_y, speed of the ball v_x, v_y and status of each brick (booleans) b_{ij}. These features of the system give all the needed informations to predict the next state from the current state. Hence we can build an MDP where: S is the set of all the possible values of the sequence of features $\langle p_x, b_x, b_y, v_x, v_y \rangle$, $A = \{\textit{right, left, no-action}\}$, transition function T determined by the rules of the game. We give reward $R(s, a, s') = 10$ if a particular brick in s' has been removed for the first time, plus 100 if that brick was the last (i.e. *environment goal* reached).

Temporal goal: The *temporal goal* (specified by a LTL$_f$/LDL$_f$ formula) is to remove lines of bricks in a given order. In other words, all bricks on each line of bricks i must be removed before removing all bricks from line $j > i$. In our experiments we considered the following goals (and combinations of them):

- BREAK-COLS-(LR|RL): remove columns of blocks from (left to right | right to left).
- BREAK-ROWS-(BT|TB): remove rows of blocks from (bottom to top | top to bottom).

For each temporal goal, we give a reward $r = 10000$ and it is given when the agent fulfilled the temporal goal specified by the formula. The formulas are expressed in LDL$_f$. For example, in Breakout 3x3 (i.e. three rows and three columns of bricks), the formula which specify the just explained temporal goals is:

$$\langle(\neg l_0 \wedge \neg l_1 \wedge \neg l_2)^*; (l_0 \wedge \neg l_1 \wedge \neg l_2); (l_0 \wedge \neg l_1 \wedge \neg l_2)^*; (l_0 \wedge l_1 \wedge \neg l_2); (l_0 \wedge l_1 \wedge \neg l_2)^*; (l_0 \wedge l_1 \wedge l_2)\rangle tt \tag{8.1}$$

where the fluent l_i means "the i_{th} line has been removed". The automaton associated to the formula in Equation 8.1 is depicted in Figure 8.1. l_i are the fluents of interests, hence $\mathcal{P} = \{l_0, l_1, l_2\}$ and $\mathcal{L} = 2^{\mathcal{P}}$.

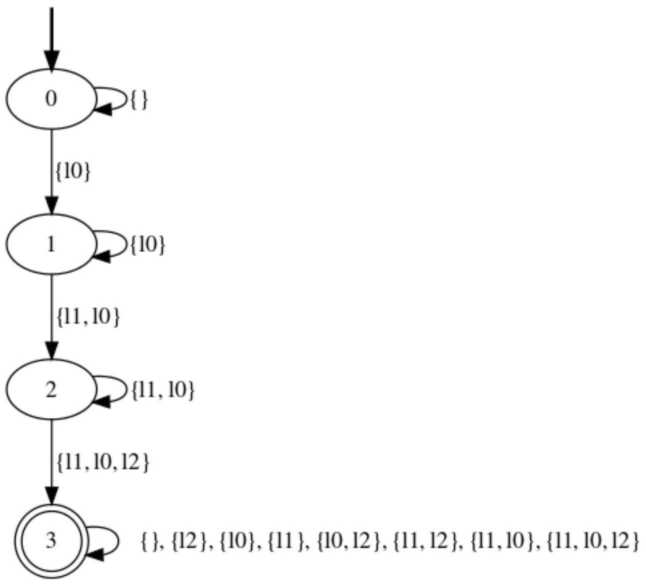

0
{}
{l0}
1
{l0}
{l1, l0}
2
{l1, l0}
{l1, l0, l2}
3
{}, {l2}, {l0}, {l1}, {l0, l2}, {l1, l2}, {l1, l0}, {l1, l0, l2}

Figure 8.1. The automaton associated to the LDL$_f$ formula in Equation 8.1 for BREAKOUT 3x3.

The features for fluents evaluation, for every temporal goal, are the status of each brick (present or removed). Recalling the notation used in Example 4.2: $\langle b_{11}, \ldots, b_{nm}\rangle$.

It is crucial to remark that the evaluation of the fluent from the features is different among the temporal goals above mentioned, despite the formulas are structurally the same (apart from the number of lines to remove). Indeed, thanks to the fluents evaluation phase (i.e. the map from features values to the truth of each fluent), the formulas behave differently.

Configurations: The reinforcement algorithm used is Sarsa(λ) (e.g. Sarsa with eligibility traces) with ϵ-greedy policy. The values of the parameters are:

- $\lambda = 0.99$
- $\gamma = 0.999$
- $\alpha = 0.1$

- $\epsilon = 0.1$

The experiments are stopped when in the last 100 episodes of optimal runs the agent always achieved both the environment goal and the temporal goals. Notice that this approach may yield a sub-optimal policy, but the found policies always satisfy LTL$_f$/LDL$_f$ goals. This method is used also for the other experiments.

8.1.1 Optimal policies

Now we show some screenshots of the optimal policies found for some of the temporal goals described before. You can find full recordings at https://www.youtube.com/channel/UChpe0QEtRSKh4uCy5f2XuGA.

Break-Cols-LR

In Figure 8.2 are depicted the highlights of a run of the optimal policy for the goal Break-Cols-LR. As you can notice, the constraint expressed by the formula in Equation 8.1 is satisfied (i.e. the columns of bricks are broken from left to right).

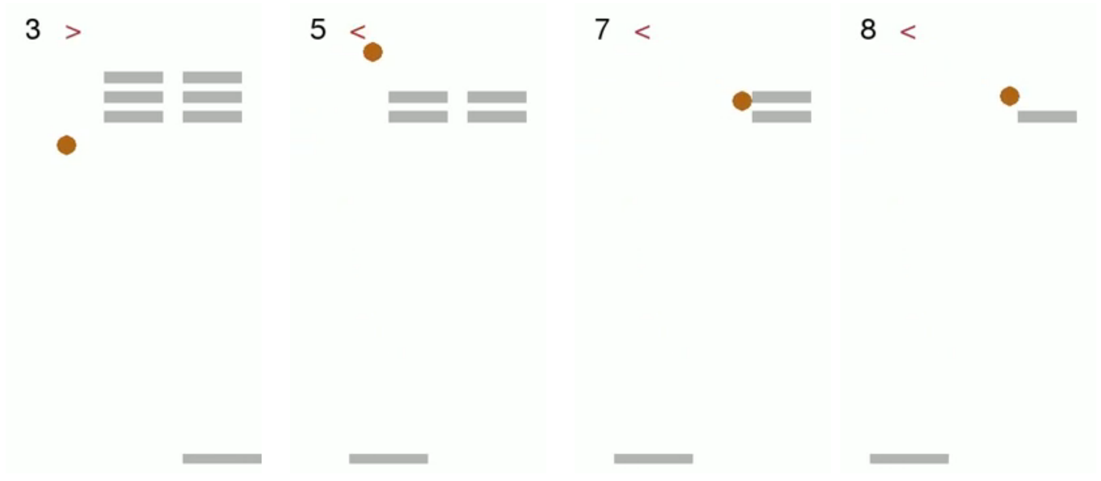


Figure 8.2. A run of the learnt optimal policy for the task Break-Cols-LR. From left to right, you can see that the columns are broken in the right order.

Break-Cols-BT

In Figure 8.3 are depicted the highlights of a run of the optimal policy for the goal Break-Cols-BT. The constraint expressed by the formula in Equation 8.1 is satisfied (i.e. the rows of bricks are broken from the bottom to the top).

Break-Cols-RL-TB

In Figure 8.4 are depicted the highlights of a run of the optimal policy for the goal Break-Cols-RL-TB. Notice that, even in the case of multiple temporal goals, the constraints expressed by the formula in Equation 8.1 are satisfied (i.e. the rows of bricks

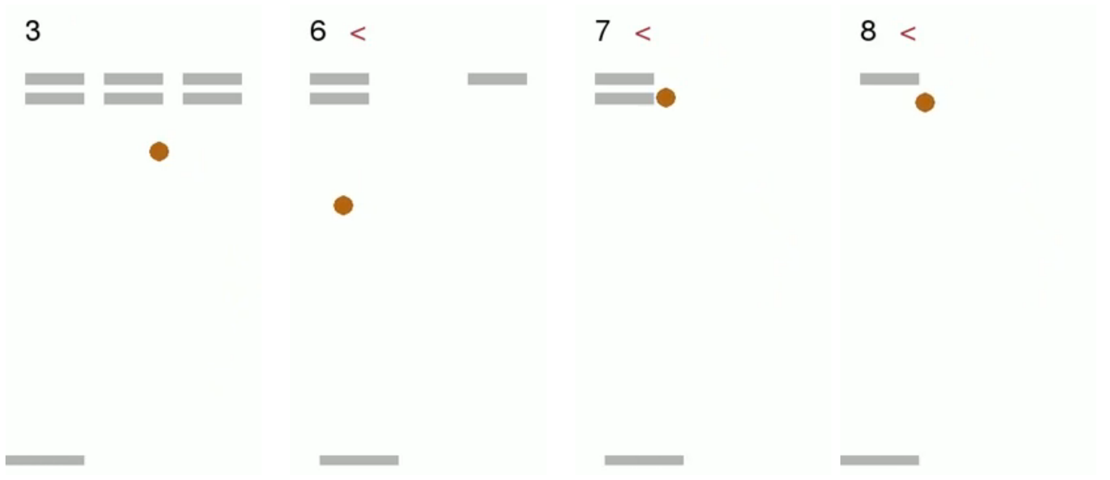


Figure 8.3. A run of the learnt optimal policy for the task Break-Rows-BT. From the bottom to the top, you can see that the rows are broken in the right order.

are broken from the top to the bottom and the columns of bricks are broken from right to left). Furthermore, notice that the formula is the same, but *how the fluents are evaluated from the features* makes the difference.

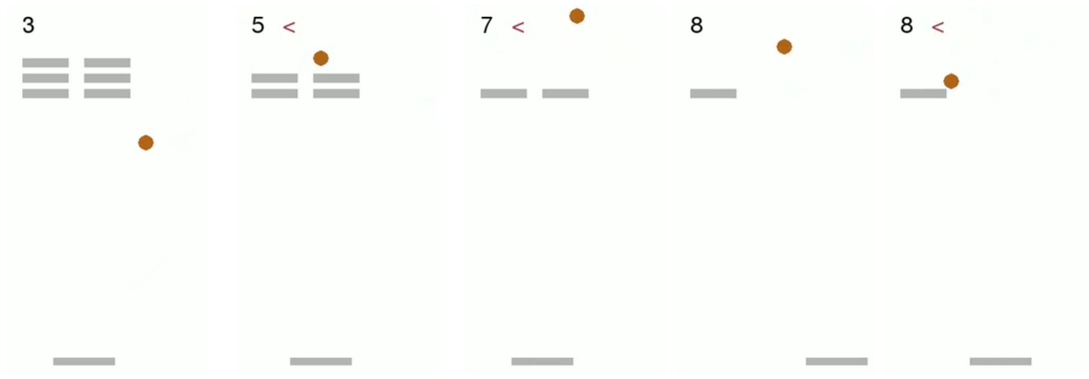


Figure 8.4. A run of the learnt optimal policy for the task Break-Cols-RL-TB. You can see that the rows are broken from the top to the bottom and that, at the same time, the columns are broken from the right to the left.

8.1.2 Benchmarking of reward shaping techniques

In this section, we report some statistics about the performances of the proposed approach with a focus on automata-based reward shaping. For each configuration, we collected the observed reward for each episode, over 10 runs. Each run has a time limit of 10000 episodes. Then we moving averaged the sequences of rewards with a window of size 100, in order to make the curves more smoothed, and averaged the result across all the runs. The plots show this processed sequences over the number of episodes. The coloured bands represent the 90% confidence interval taken from bootstrapped resamples.

We show our results in two parts:

1. Comparison of *No Reward Shaping*, *Off-line Reward Shaping* and *On-the-fly Reward Shaping* in BREAKOUT 3x3, 3x4 and 4x4, with temporal goal BREAK-COLS-LR;

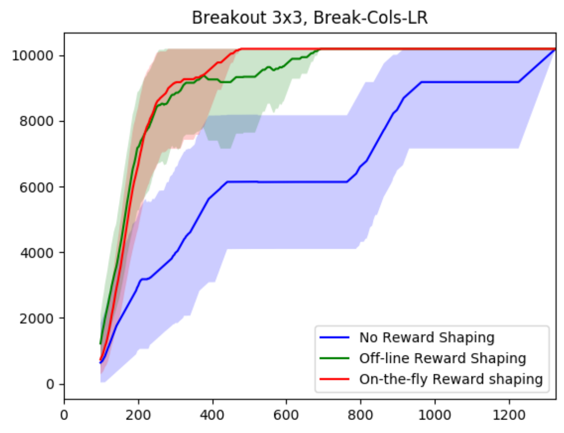
2. Comparison of *No Reward Shaping*, *Off-line Reward Shaping* and *On-the-fly Reward Shaping* in Breakout 4x4, with temporal goals BREAK-COLS-LR, BREAK-ROWS-BT, BREAK-COLS-LR & BREAK-ROWS-BT;

Different number of bricks and rows

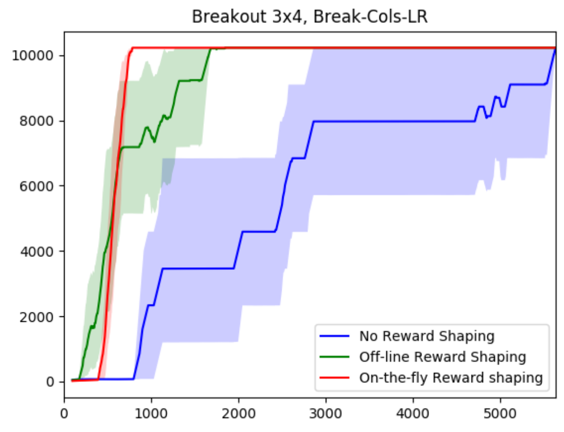
In Figure 8.5 are plotted different statistics by increasing difficulty in the environment, for every variant of reward shaping technique. On the x-axis the number of episodes, on the y-axis the average reward (obtained as explained above). In every case, we can see that the use of reward shaping (both off-line and on-the-fly) speed-up the learning process.

Different temporal goals

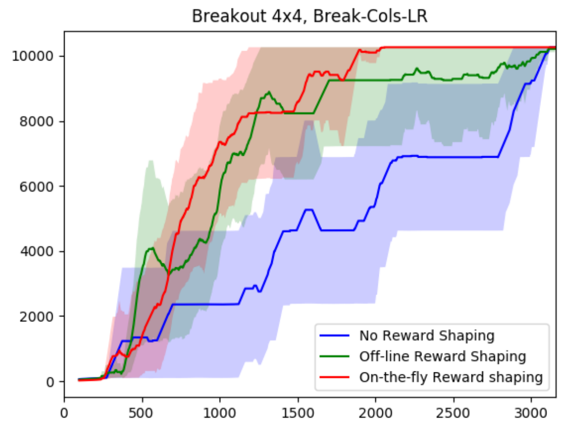
In Figure 8.6 Here we show the performances by varying the temporal goals. On the x-axis the number of episodes, on the y-axis the average reward (obtained as explained above). In every case, we can see that the use of reward shaping (both off-line and on-the-fly) speed-up the learning process.



(a) BREAKOUT 3x3, BREAK-COLS-LR for three different settings: *No RS*, *Off-line RS* and *OTF RS*

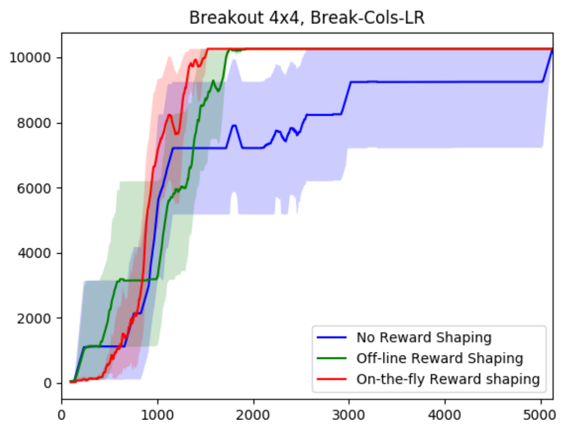


(b) BREAKOUT 3x4, BREAK-COLS-LR for three different settings: *No RS*, *Off-line RS* and *OTF RS*

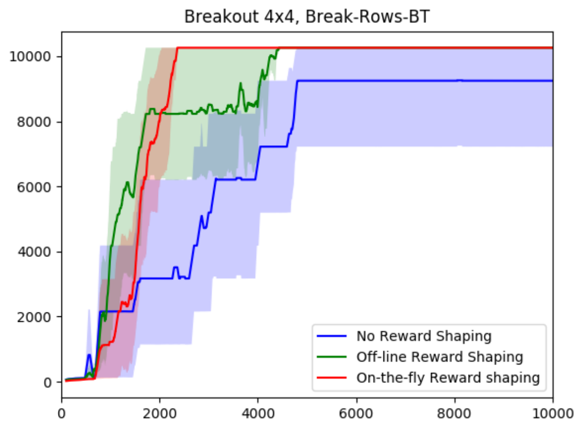


(c) BREAKOUT 4x4, BREAK-COLS-LR for three different settings: *No RS*, *Off-line RS* and *OTF RS*

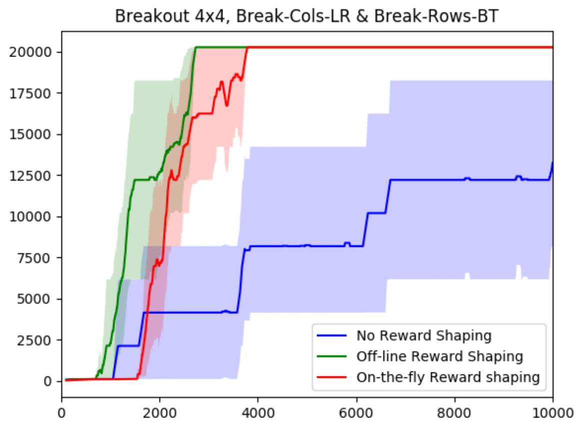
Figure 8.5. The results of three settings are reported, namely: BREAKOUT 3x3, 3x4 and 4x4 (8.5a, 8.5b and 8.5c respectively), all of them with temporal goal BREAK-COLS-LR. On the x-axis the episodes, on the y-axis the average reward. In every case, we can see that the use of reward shaping (both off-line and on-the-fly) speed-up the learning process.



(a) BREAKOUT 4x4, BREAK-COLS-LR for three different RS modes.



(b) BREAKOUT 4x4, BREAK-ROWS-BT for three different RS modes.



(c) BREAKOUT 4x4, BREAK-COLS-LR & BREAK-ROWS-LR for three different RS modes.

Figure 8.6. The results of three settings are reported, namely: BREAKOUT 4x4 with temporal goal BREAK-COLS-LR, BREAK-COLS-BT and BREAK-COLS-LR & BREAK-COLS-BT (8.6a, 8.6b and 8.6c respectively). Also in this case, the use of reward shaping speed-up the learning process.

8.2 SAPIENTINO

SAPIENTINO Doc is an educational game for 5-8 y.o. children in which a small mobile robot has to be programmed in order to visit specific cells in a 5x7 grid. Cells contain concepts that must be matched by the children (e.g., a coloured animal, a colour, and the initial letter of the name of the animal). The robot executes sequences of actions given in input by children with a keyboard on the robot's top side. During execution, the robot moves on the grid and executes an action (actually a *bip*) to announce that the current cell has been reached (this is called a *visit* of a cell). In this paper, we generalize this game as follows. As in the real game, we consider a 5x7 grid with 7 triplets of coloured cells, each triplet representing three matching concepts (see Figure 8.7 for a screenshot).

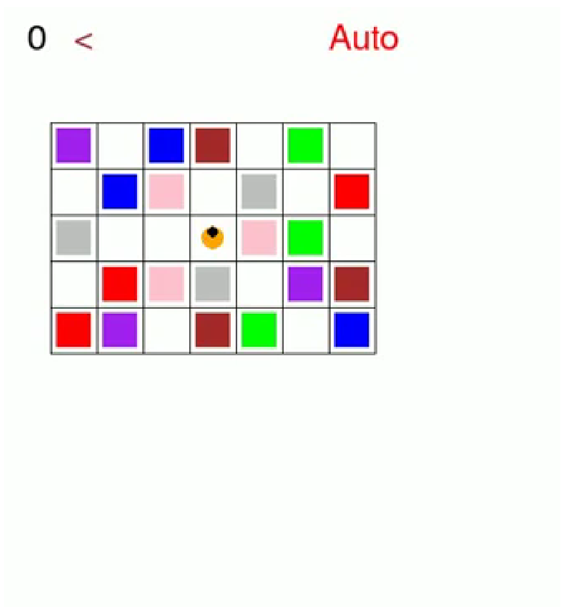


Figure 8.7. A screenshot from SAPIENTINO

Actions, Features and Fluents: The actions available to the agent are: UP, DOWN, LEFT, RIGHT and BIP. The features for the agent space are the position (x, y) of the agent in the grid, namely f_x and f_y. The features for evaluating the fluents configurations are: f_b reporting if a BIP has just been executed, and f_c denoting the colour of the current cell. In our setting, the available fluents are:

$$\mathcal{P} = \{bip, red, green, blue, pink, brown, grey, purple\}$$

with the obvious meaning that their name indicates (just map features to fluents one-to-one).

Temporal Goal: In our experiment, we considered the following temporal behaviours:

- ORDERED-VISITS: *visit* each colour and do a *bip* in each colour in a given order.

- ILLEGAL-BIPS: between each visit, never do a *bip*.

We identified two temporal goals: one considering that both conditions must be satisfied and a more relaxed one that only ORDERED-VISITS must be satisfied. They can be translated, respectively, into the following LDL$_f$ formulas:

$$\begin{aligned}&\langle(\neg bip)^*; red \wedge bip; (\neg bip)^*; green \wedge bip;\\ &(\neg bip)^*; blue \wedge bip; (\neg bip)^*; pink \wedge bip;\\ &(\neg bip)^*; brown \wedge bip; (\neg bip)^*; grey \wedge bip;\\ &(\neg bip)^*; purple \wedge bip\rangle tt\end{aligned} \tag{8.2}$$

and:

$$\begin{aligned}&\langle true^*; red \wedge bip; true^*; green \wedge bip;\\ &true^*; blue \wedge bip; true^*; pink \wedge bip;\\ &true^*; brown \wedge bip; true^*; grey \wedge bip;\\ &true^*; purple \wedge bip\rangle tt\end{aligned} \tag{8.3}$$

where Formula 8.3 is the same of Formula 8.2 but replacing every occurrence of $(\neg bip)^*$ with $true^*$. The associated automaton are depicted, respectively, in Figure 8.8a and 8.8b.

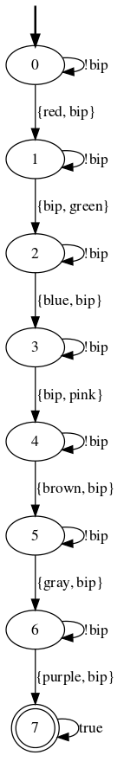
Notice that the condition of illegal bips does not affect the optimal policy; on the other hand, it highly affects the exploration phase, since with the ILLEGAL-BIPS condition, every time the ϵ-greedy policy chooses *bip* randomly and "illegally", then that trajectory cannot satisfy the temporal goal anymore. Moreover, observe that, in this case, we do not consider any *environment goal*, as we did in BREAKOUT experiments (i.e. break all the bricks).

Configurations: The reinforcement algorithm used is Sarsa(λ) (e.g. Sarsa with eligibility traces) with ϵ-greedy policy. The values of the parameters are:

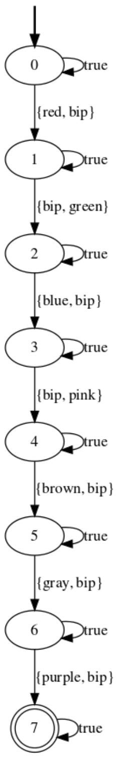
- $\lambda = 0$
- $\gamma = 1.0$
- $\alpha = 0.1$
- $\epsilon = 0.1$

We show our results in two parts:

1. Comparison of *No Reward Shaping*, *Off-line Reward Shaping* and *On-the-fly Reward Shaping* in SAPIENTINO with the "full" temporal goal, i.e. ORDERED-VISITS & ILLEGAL-BIPS;
2. Comparison of *No Reward Shaping*, *Off-line Reward Shaping* and *On-the-fly Reward Shaping* in SAPIENTINO with the "relaxed" temporal goal, i.e. only ORDERED-VISITS ;



(a) The automaton associated to the LDL$_f$ formula in Equation 8.2 for for the "full" temporal goal in SAPIENTINO.



(b) The automaton associated to the LDL$_f$ formula in Equation 8.3 for the "relaxed" temporal goal in SAPIENTINO.

We analyze the results of Experiment 1 and 2. As stated before, since the two experiments differ only for the exploration phase, the optimal policy that satisfies the goal is qualitatively the same. You might see the resulting learnt policy at `https://www.youtube.com/watch?v=ghBImtYMpVk`.

8.2.1 Ordered-Visits & Illegal-Bips

In this experiment, we observed that the agent successfully learned the temporal goal, both in off-line and in on-the-fly reward shaping. The performances are pretty the same. Notice that, without reward shaping, in our simulations (5000 episodes), *the agent never reached the goal*. This is due mainly to the long sequence of actions that the agent has to take in order to accomplish the entire goal, considering that an illegal *bip* lead to a failure of the goal and so the end of the episode. On the other hand, in the case of reward shaping, an illegal bip is punished because the episode ends and the agent collects the negative potential fall. Hence, the agent recognizes, earlier than no reward shaping configuration, that the illegal bip is a bad action.

8.2.2 Ordered-Visits

Also in this case, the agent learned the temporal goal, even in the case where no reward shaping was applied. It is interesting to note that a small change in the formula can affect drastically the learning process, due to exploration phase constraints.

8.3 MINECRAFT

MINECRAFT (Andreas et al., 2017) is a sort of 2D porting of the well-known 3D videogame Minecraft. The agent can *get* resources (e.g. *wood*, *grass*) and *use* tools (e.g. *toolshed*, *workbench*) located on a grid (similarly to the grid in SAPIENTINO), and has to accomplish composite tasks. For instance, the task *make a bed* is divided into the following sequence of subtasks: *get_wood*, *use_toolshed*, *get_grass*, *used_workbench*. In Figure 8.9 you can see a screenshot of the game.

Actions, Features and Fluents: The actions available to the agent are: UP, DOWN, LEFT, RIGHT, GET and USE. The features for the agent space are the position (x, y) of the agent in the grid, namely f_x and f_y. The features for evaluate the fluents configurations are: f_g reporting if a GET has just been executed, f_u reporting if a USE has just been executed, f_ℓ denoting the resource/tool available in the current location, and f'_ℓ, if a resource has been taken and is available for the incoming tasks. In our setting, the available fluents are:

$$\mathcal{P} = \{get\_wood, get\_grass, get\_iron, use\_toolshed, use\_workbench, use\_factory\}$$

when the agent is near a resource (i.e. *wood*, *grass* or *iron*) and do a GET, then the associated "get" fluent becomes true. Analogously for tools and the USE action. When a task is completed and to do that some previously collected resource/tool has been used, then the resources are lost and has to be recollected again in order to accomplish other tasks.

Temporal Goal: In our experiment, we considered the following tasks:

1. MAKE-BED: $get\_wood, use\_toolshed, get\_grass, use\_workbench$
2. MAKE-AXE: $get\_wood, use\_workbench, get\_iron, use\_toolshed$
3. MAKE-BRIDGE: $get\_iron, get\_wood, use\_factory$

Now we show how to express them in LDL$_f$ and its associated automaton. For simplicity, we show only the ones for the goal 3. The associated LDL$_f$ formula is:

$$\begin{aligned}&\langle true^*\rangle\langle get\_iron \land \neg get\_wood \land \neg use\_factory;\\ &\quad (get\_iron \land \neg get\_wood \land \neg use\_factory)^*; get\_iron \land get\_wood \land \neg use\_factory;\\ &\quad (get\_iron \land get\_wood \land \neg use\_factory)^*; get\_iron \land get\_wood \land use\_factory\rangle tt\end{aligned} \tag{8.4}$$

whereas the translation of Formula 8.4 into automaton is depicted in Figure 8.10:

Notice that we could "relax" the specified order of *get_wood* and *get_iron*, since it is not relevant to the operation *make a bridge*.

The agent is trained to satisfy all the three tasks in a single episode. For each temporal goal, the reward assigned is 1.

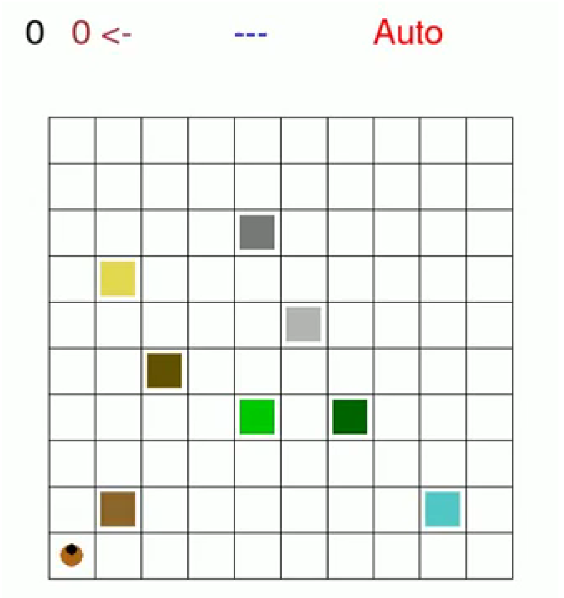


Figure 8.9. A screenshot of the game MINECRAFT

Configurations: The reinforcement algorithm used is Sarsa(λ) (e.g. Sarsa with eligibility traces) with ϵ-greedy policy. The values of the parameters are:

- $\lambda = 0.9$
- $\gamma = 0.99$
- $\alpha = 0.1$
- $\epsilon = 0.25$

8.3.1 Results

In Figure 8.11 is plotted the benchmarking, similarly to the one presented in Section 8.1.2. We notice that the policy that satisfies every temporal goal has been found when a reward shaping is applied, whereas in the case of no reward shaping the time limit imposed for the experiment was not enough to allow every run to achieve the goal. You can see one of these found policies at `https://youtu.be/IJ3Hr79xfBs`.

Observe that reward shaping, both in off-line and on-the-fly variant, outperform the absence of reward shaping. This is due mainly to the high number of steps that the agent has to make in a precise order in order to achieve the goal: without a guide in the exploration (e.g. shaping rewards) it is hard to discover the proper sequence that leads to the satisfaction of every formula. Furthermore, observe that off-line reward shaping is slightly better than on-the-fly reward shaping in terms of convergence rate. It is also more stable, as the reader might infer from the shaded regions that represent the variance of the plot: indeed, it is not surprising that on-the-fly reward shaping yields a more noisy learning process, due to the dynamic change of the potential function and so of the shaping rewards.

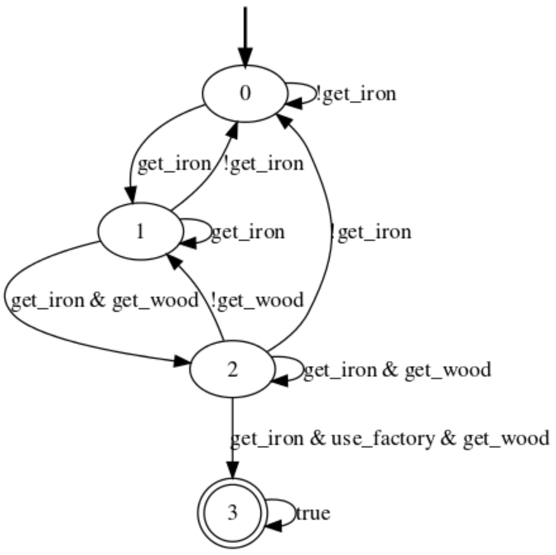


Figure 8.10. An approximate representation of the automaton associated to Formula 8.4. Notice that it is "simplified" by omitting groups of transitions that are collapsed in a single edge labeled with the propositional formula that the interpretation should satisfy.

It is worth to notice the particular structure of the environment in our setting. Indeed, when one of the tasks is completed, the resources that progressed at the same time the other automata/tasks are released; this lead to a regression of the state of those tasks, and so to a negative shaping reward. Hence, the positive shaping reward given for progression is mitigated by the negative shaping reward due to completion of a task and regression of the other task that has some resource in common. The learning agent has to deal with this issue, that at the scaling of the problem it might lead to slow the learning.

8.4 Implementation

The experiments are implemented using FLLOAT and RLTG, presented in Chapter 3 and Chapter 7. The code for running the experiments by yourself with other configurations and plot the statistics are published in this repository. You'll find also the scripts to run the experiments analyzed in this chapter. The BREAKOUT, SAPIENTINO and MINECRAFT implementations can be found in this repository, which is a fork of this one, created by prof. Luca Iocchi.

8.5 Conclusion

In this chapter we've seen practical examples and implementations of the topics described in the previous chapters, by using the implementation described in Chapter 3 and 7.

We first used the BREAKOUT environment and introduced some LTL$_f$/LDL$_f$ goals to

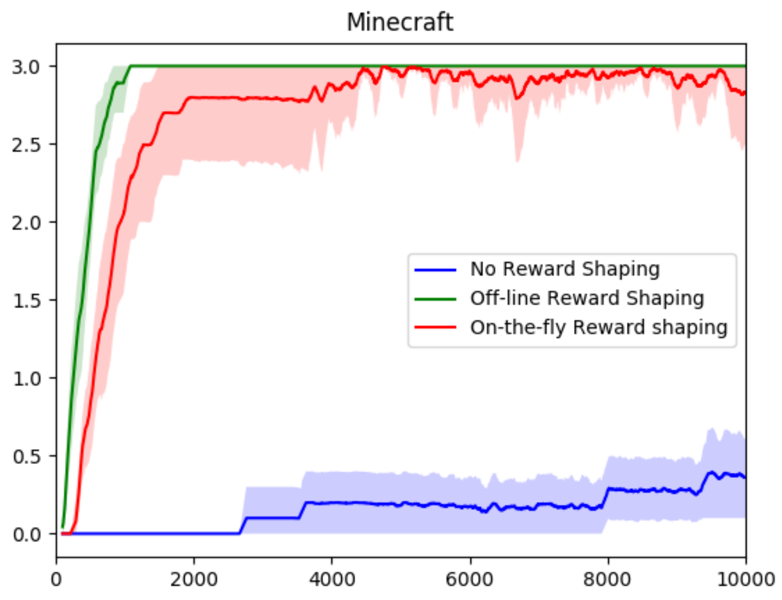


Figure 8.11. A comparison between reward shaping techniques for the temporal goals 1, 2 and 3. On the x-axis the episodes, on the y-axis the average reward, as explained in Section 8.1.2.

satisfy. We noticed that, in every case, the optimal policies have been learnt, even in the case of multiple constraints (i.e. remove both columns and rows in a given order). Moreover, we provided experimental evidence of the better performances achieved in the presence of reward shaping (both variants) with respect to no reward shaping.

Then we moved to SAPIENTINO, where the agent has to visit some cells on a grid in a given order. The result is that with reward shaping the agent always learnt the optimal policy, whereas with sparse rewards the agent did not succeeded, as another evidence of the importance of reward shaping. Moreover, we emphasized some issues in the specification of the formula, that despite does not affect the optimal policy, it highly affects the exploration phase.

Our last experiment uses MINECRAFT, where the agent has to accomplish composite tasks. We set up the experiment with 3 of those tasks, where some of them are interleaved, i.e. require the same resources/tools. We've shown that also in this case the agent learnt the optimal policy, and that the reward shaping techniques outperform the absence of reward shaping.

Chapter 9

Conclusion and Future Work

This chapter summarizes the thesis. It starts with a brief overview of the problem studied. Then we summarize the main contributions of the thesis. A section is dedicated to future directions of research. The document ends with final remarks.

9.1 Overview

This work addressed the problem of Reinforcement Learning in the context of NMRDP, i.e. when the rewards are non-Markovian and so they depend on a sequence of transitions. In the literature, the problem has been solved exclusively in the planning domain, by two steps. First, by expressing the temporal goals in a temporal logic formalism (e.g. PLTL, \$FLTL, LTL$_f$) and so by implicitly denoting the trajectories to be rewarded. Then, by devising an expansion of the original state space such that each primitive state is copied multiple times in order to capture the relevant histories for the satisfaction of the temporal goals, and making the transition model to satisfy the Markov property. This transformation into an expanded MDP allows to run classic reinforcement learning algorithms and so to implicitly learn the non-Markovian policy to solve the original problem.

Here, instead, we focused on doing Reinforcement Learning for NMRDPs. We used the definition of the goals in NMRDP by using LDL$_f$ formalism, that is expressive as MSO logic. The transformation to an equivalent MDP is provided by results in (Brafman et al., 2018). Moreover, we consider the case when can be useful to devise two different representations of the world. We considered a classical reinforcement learning problem, i.e. an MDP over a set of features of the world, that we called *low-level* features [1]. We are interested in defining non-Markovian rewards over this MDP about some high-level properties of the environment, that we called the *fluents*, determined by some set of *high-level* features of the world. This approach gives us several advantages, such as modularity of the system, learning over a reduced state space (minimal in terms of what is needed for the agent) and declarativeness of the target behaviours in terms of the fluents.

[1]We say *low-level* features in the sense that these features are only responsible for the basic interaction of the agent with the environment, but still allowing to maximize the reward, or reach the goal in the MDP. Notice that it is only a way to refer about them, and no restriction is made over the MDP state space.

Moreover, we are interested in an implementation of these topics, by providing software that deals with the specification of LTL$_f$/LDL$_f$ formulas, the computation of the extended state space and the running of reinforcement learning algorithms that solve these problems.

9.2 Main Contributions

In this section, we list the main contribution of the thesis.

- Explanation of the transformation from LTL$_f$/LDL$_f$ formulas to automata, supported by several examples in every variant. Design of a new algorithm for the translation from LTL$_f$/LDL$_f$ formulas to DFA (LDL$_f$2DFA, Algorithm 2.2), for implementation purposes. Actually, it is a variant of the LDL$_f$2NFA algorithm described in (Brafman et al., 2018).

- Implementation of the Python package FLLOAT (From LTL$_f$ and LDL$_f$ tO auTomata). We described the main features and several use cases in Chapter 3. It supports the representation of PL and LTL$_f$/LDL$_f$ formulas, as well as their truth evaluation. As a core feature, it implements the transformation from LTL$_f$/LDL$_f$ formulas to equivalent automata, in the variant of LDL$_f$2NFA, LDL$_f$2DFA and on-the-fly automaton.

- As a theoretical contribution, we defined the problem of *Reinforcement Learning over NMRDP with* LTL$_f$/LDL$_f$ *rewards*, by leveraging the results in (Brafman et al., 2018). Indeed, the proposed solution reduces the problem to classic reinforcement learning over an equivalent MDP with an expanded state space. Since an optimal policy for the extended MDP can be transformed to an optimal policy for NMRDP, the problem reduces to find an optimal policy for the MDP.

- A relevant part of this work dealt with a two-fold representation of the world. One is the agent's representation that the agent uses to learn. The other one is used for specifying temporally extended goals expressed in LTL$_f$/LDL$_f$. We formally defined the problem, that we call *RL for* LTL$_f$/LDL$_f$ *goals*, and provide a solution. The idea is to reduce the new problem to an instance of *RL over NMRDP with* LTL$_f$/LDL$_f$ *rewards*, which in turn can be reduced to plain RL over MDP. We formally describe our approach step by step.

- We devised a way to apply *reward shaping* to this setting, by leveraging the particular structure of our solution. Indeed, the core observation is the following: the transitions that make a step toward the satisfaction of the LTL$_f$/LDL$_f$ goals can be seen as transitions that make a step toward an accepting state of $\mathcal{A}_\varphi$. Hence, by positively rewarding this transitions (and negatively the opposite ones), we help the agent to explore the extended state space, preventing the main issues caused by *sparse rewards*.

  We design the reward shaping by considering the definition of *potential based reward function* (Ng et al., 1999) and its property of policy invariance, i.e. the optimal policy does not change when applied this kind of reward shaping. The potential function associates each state of the automaton to a real number. The nearer the state to an accepting state, the higher the potential function evaluated in that state. We formally devised a procedure that, given $\mathcal{A}_\varphi$, returns a potential function defined over the states of $\mathcal{A}_\varphi$ (Algorithm 6.1) We call it *static reward shaping* or *off-line reward shaping*.

We proposed a variant of off-line reward shaping that does not require $\mathcal{A}_\varphi$. In this approach, $\mathcal{A}_\varphi$ is built from scratch, during the learning process, by observing the transitions and collecting information about the states of the automata and the value of the fluents that allow the transition. We call this approach *on-the-fly* reward shaping.

- We implemented RLTG (Reinforcement Learning for Temporal Goal), a Python framework for easy set up a reinforcement learning experiment with LTL$_f$/LDL$_f$ goals, by leveraging the above-mentioned FLLOAT for the construction of the automata. It works both as a classic reinforcement learning framework and a LTL$_f$/LDL$_f$ goal-based framework, supporting the settings presented in Chapter 4 and Chapter 5, as well as the reward shaping techniques explained in Chapter 6.

  We achieved an effective modularization of a reinforcement learning system, that allows the specification of temporal goals over a set of fluents, whose truth value is extracted from the RL environment by user-defined functions. We aimed to highly customizability of the components: features extraction of the features set of the agent and of the LTL$_f$/LDL$_f$ goals, mapping from features to fluents, the specification of LTL$_f$/LDL$_f$ formulas to be satisfied by the agent.

  The available RL algorithms are Q(λ) and Sarsa(λ), but the framework allows to easily implement new algorithms and seamlessly integrate them with the other components of the framework. It allows the user to pause the learning process in order to resume it later, to render the agent while simulating in the environment and to collect statistics about the learning.

- Finally, we designed experiments with simulated RL environments and implemented them with FLLOAT and RLTG. For each experiment, we specified a temporal goal with a LTL$_f$/LDL$_f$ formula.

  We started from a classic RL environment, BREAKOUT, and introduced a temporally extended goal (break columns/rows in a given order). We made the following observations: we noticed that the agent learnt the optimal policies in many different version of the temporal goal (e.g. break columns from left-to-right/right-to-left and break rows from top-to-bottom/bottom-to-top), even when considered both of them (but not mutually exclusive ones). Moreover, we compared the use of off-line reward shaping, on-the-fly reward shaping and any use of reward shaping, and we observed that the use of reward shaping outperformed the configuration without reward shaping.

  Then we moved to the SAPIENTINO environment, and we specified two versions of the goal: one that the agent had to visit the colors of the grid in a given order *relaxed*, and another, more restricting, that requires also that no bip action can be executed if the agent is not over a colored cell *full*. For the relaxed temporal goal, the agent always learnt the optimal policy in a limited amount of time. In the full temporal goal, only in the configurations with reward shaping the agent actually accomplished the goal, which had not happened in the configuration without reward shaping. These experiments show that the reward shaping is necessary for a complex temporal goal to be learnt in a reasonable amount of time and that a slightly different temporal specification might highly affect the exploration phase.

The last experimented environment is MINECRAFT, where the agent had to accomplish many tasks, each of them composed by many subtasks that might be shared among other tasks. We run the experiment with three tasks, and compared the different use of reward shaping, as we did in the BREAKOUT experiment. The result is that, still, the reward shaping configurations outperform the no-reward shaping configurations, in terms of the time needed to learn every temporal goal. Moreover, we noticed that the particular structure of the temporal goals might have introduced difficulties in the learning, due to the interleaving of different tasks. However, the agent has been able to learn a proper policy in a reasonable amount of time.

9.3 Future Works

There are many future directions that can be taken, due to the novelty of the work. Some of them are:

- It could be interesting to find out the correlation between the low-level representation and the high-level representation that allows the agent to learn how to accomplish the goals. In particular, it is interesting to find how to build a *minimal* low-level representation, from the high-level one, that permits the satisfaction of the LTL_f/LDL_f formulas. The mentioned finding could help the design phase of the relevant features of the learning agent.

- FLLOAT and RLTG, the implementations of the theoretical topics of this work, introduced in Chapter 3 and Chapter 7, can be improved in many aspects. FLLOAT can be optimized by implementing ad-hoc procedures for the conversion into automata, based on qualitative inspections of the formulas. It is possible to devise a better design and suited data structures for the problem. The RLTG framework can be enhanced by providing support for many other RL algorithms and making it easily usable while keeping its customizability.

- In this work, we prevalently made a theoretical analysis of the problem and we have shown only some application at the software level. The proposed framework is more general, and should be validated in many other domains, both simulated and physical ones (e.g. robotics), where there is the need to represent additional high-level knowledge to express complex goals;

- We did not focus our attention over the algorithmic side, but only on how the problem can be properly redefined, and relying on off-the-shelf reinforcement learning algorithms. It might be the case that the design of ad-hoc algorithms for this framework yields better performances. For instance, one could think about a better "explicit" exploration policy of the state space, in spite of "implicit" guidance by using reward shaping, and even some adaptation of state-of-the-art techniques, e.g. hierarchical reinforcement learning, curriculum learning, knowledge transfer and so on.

- Finally, it could be of interest to extend this work to the general framework of multi-agent systems and work on the challenges and issues that such extension might lead to.

9.4 Final Remarks

The topic of this work is of crucial importance: the need to face the dichotomy between the control of low-level features and the ability to reason about higher-level properties of the world is the core issue in many of the fields and applications in artificial intelligence, especially in robotics. This thesis aimed to address this issue in a particular case, by a theoretical analysis, but providing actual implementations as support for the proposed approach.

Bibliography

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 166–175, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/andreas17a.html.

Fahiem Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 1160–1167, 1996.

Jorge A. Baier, Christian Fritz, Meghyn Bienvenu, and Sheila A McIlraith. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1509–1512. AAAI Press, 2008. ISBN 978-1-57735-368-3. URL http://dl.acm.org/citation.cfm?id=1620270.1620321.

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957. URL http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false.

Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. Ltlf/ldlf non-markovian rewards, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17342.

Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. Specifying non-markovian rewards in mdps using ldl on finite traces (preliminary version). *CoRR*, abs/1706.08100, 2017.

Richard J. Büchi. Weak secondâĂŘorder arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1âĂŘ6):66–92, 1960. doi: 10.1002/malq.19600060105. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19600060105.

Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A. McIlraith. Non-markovian rewards expressed in LTL: guiding search via reward shaping. In *SOC*, pages 159–160, 2017a.

Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A. McIlraith. Decision-making with non-markovian rewards: From ltl to automata-based reward shaping. In *RLDM*, pages 279–283, 2017b.

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, volume 13, pages 854–860, 2013.

Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for ltl and ldl on finite traces. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1558–1564. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL http://dl.acm.org/citation.cfm?id=2832415.2832466.

Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on ltl on finite traces: Insensitivity to infiniteness. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 1027–1033. AAAI Press, 2014. URL http://dl.acm.org/citation.cfm?id=2893873.2894033.

Sam Devlin and Daniel Kudenko. Dynamic potential-based reward shaping. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12, pages 433–440, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9817381-1-7, 978-0-9817381-1-6. URL http://dl.acm.org/citation.cfm?id=2343576.2343638.

Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *In Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126. Morgan Kaufmann, 1998.

Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961. ISSN 00029947. URL http://www.jstor.org/stable/1993511.

Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194 – 211, 1979. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(79)90046-1. URL http://www.sciencedirect.com/science/article/pii/0022000079900461.

D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. Technical report, Jerusalem, Israel, Israel, 1997.

Valentin Goranko and Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2015 edition, 2015.

Charles Gretton. A more expressive behavioral logic for decision-theoretic planning. In *Pacific Rim International Conference on Artificial Intelligence*, pages 13–25. Springer, 2014.

M. Grzes and D. Kudenko. Theoretical and empirical analysis of reward shaping in reinforcement learning. In *2009 International Conference on Machine Learning and Applications*, pages 337–344, Dec 2009. doi: 10.1109/ICMLA.2009.33.

Marek Grzes. *Improving exploration in reinforcement learning through domain knowledge and parameter analysis*. PhD thesis, University of York, 2010.

Marek Grześ. Reward shaping in episodic reinforcement learning. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 565–573, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems. URL http://dl.acm.org/citation.cfm?id=3091125.3091208.

John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000. ISBN 0201441241.

Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Teaching multiple tasks to an rl agent using ltl. 2018.

Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and Its Applications*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 2001. ISBN 3764342072.

Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001. ISSN 0925-9856. doi: 10.1023/A:1011254632723. URL https://doi.org/10.1023/A:1011254632723.

Bruno Lacerda, David Parker, and Nick Hawes. Optimal policy generation for partially satisfiable co-safe ltl specifications. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1587–1593. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL http://dl.acm.org/citation.cfm?id=2832415.2832470.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-612-2. URL http://dl.acm.org/citation.cfm?id=645528.657613.

M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Proceedings of the 2006 International Conference on Business Process Management Workshops*, BPM'06, pages 169–180, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-38444-8, 978-3-540-38444-1. doi: 10.1007/11837862_18. URL http://dx.doi.org/10.1007/11837862_18.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32.

M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959. ISSN 0018-8646. doi: 10.1147/rd.32.0114. URL http://dx.doi.org/10.1147/rd.32.0114.

Stewart Shapiro and Teresa Kouri Kissel. Classical logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2018 edition, 2018.

Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Mach. Learn.*, 22(1-3):123–158, January 1996. ISSN 0885-6125. doi: 10.1007/BF00114726. URL http://dx.doi.org/10.1007/BF00114726.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3837. URL http://doi.acm.org/10.1145/3828.3837.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, Aug 1988. ISSN 1573-0565. doi: 10.1007/BF00115009. URL https://doi.org/10.1007/BF00115009.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Sylvie Thiébaux, Charles Gretton, John K. Slaney, David Price, and Froduald Kabanza. Decision-theoretic planning with non-markovian rewards. *J. Artif. Intell. Res. (JAIR)*, 25:17–74, 2006.

Wolfgang Thomas. Star-free regular sets of ÏĽ-sequences. *Information and Control*, 42(2):148 – 156, 1979. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(79)90629-6. URL http://www.sciencedirect.com/science/article/pii/S0019995879906296.

B.A. Trakhtenbrot. Finite automata and the logic of single-place predicates. *Sov. Phys., Dokl.*, 6:753–755, 1961. ISSN 0038-5689.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, 1989.

Pierre Wolper. Temporal logic can be more expressive. *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pages 340–348, 1981.