Certified Reinforcement Learning with Logic Guidance

Mohammadhosein Hasanbeig
Alessandro Abate
Daniel Kroening

Abstract This paper proposes the first model-free Reinforcement Learning (RL) framework to synthesise policies for an unknown, and possibly continuous-state, Markov Decision Process (MDP), such that a given linear temporal property is satisfied. We convert the given property into a Limit Deterministic Büchi Automaton (LDBA), namely a finite-state machine expressing the property. Exploiting the structure of the LDBA, we shape an adaptive reward function on-the-fly, so that an RL algorithm can synthesise a policy resulting in traces that probabilistically satisfy the linear temporal property. This probability (certificate) is also calculated in parallel with learning, i.e. the RL algorithm produces a policy that is certifiably safe with respect to the property. Under the assumption that the MDP has finite number of states, theoretical guarantees are provided on the convergence of the RL algorithm. We also show that our method produces "best available" control policies when the logical property cannot be satisfied. Whenever the MDP has a continuous state space, we empirically show that our framework finds satisfying policies, if there exist such policies. Additionally, the proposed algorithm can handle time-varying periodic environments. The performance of the proposed architecture is evaluated via a set of numerical examples and benchmarks, where we observe an improvement of one order of magnitude in the number of iterations required for the policy synthesis, compared to existing approaches whenever available.

1 Introduction

Markov Decision Processes (MDPs) are a family of stochastic processes adopted in automatic control, computer science, economics, and biology inter alia, for modelling sequential decision-making problems [1]. By choosing relevant actions over states, a decision maker (or an agent) can move probabilistically between states [2] and receives a scalar reward. The outcomes from taking actions are in

Department of Computer Science, University of Oxford.
E-mail: {hosein.hasanbeig, alessandro.abate, daniel.kroening}@cs.ox.ac.uk.

general probabilistic and not fully under the control of the agent [3]. An MDP is said to be solved when at any given state the agent is able to choose most favourable actions so that the accrued reward is expected to be maximum in the long run: in other words, the goal is to find an optimal action selection policy that returns the maximum expected reward [2] (possibly discounted over time).

When state and action spaces are finite, the stochastic behaviour of the MDP is encompassed by a transition probability matrix, which represents its mathematical model. In problems where this matrix is available, the most immediate method to solve a given MDP is to use Dynamic Programming (DP). DP iteratively applies a Bellman operation on a value function expressing the expected reward of interest, which is defined over the "entire state space" of the MDP [3]. Due to its reliance on the whole state space and its known computational costs, the applicability of DP can be practically quite limited. When the state and action spaces are not finite, approximate DP is often employed. This approximation can be applied over the state or action space [4–7], or over the value function [8–10].

In practice, holding full knowledge about the stochastic behaviour of the MDP (namely, its model) is often not feasible. Consider a robotic motion planning problem as an example: at a given state taking a particular action might move the robot to a different state each time due to several factors both in the environment where the robot operates and also in the mechanics of the robot; thus, the robot planning problem can be modelled as an MDP in which the state are observable but transition probabilities are unknown. In these scenarios classical DP is of limited utility, because of its assumption of a perfect model [11].

Reinforcement Learning (RL), on the other hand, is an algorithm that is widely used to train an agent to interact with an unknown MDP. RL is inspired by cognitive and behavioural psychology, where a reinforcement is the outcome of an action that will strengthen an agent's future behaviour, whenever that behaviour is anticipated by a specific stimulus. A key feature of (model-free) RL is its sole dependence on these set of experiences, which, in the form of traces, have a time sequentiality. This makes RL inherently different than DP, in the sense that it can solve an MDP without having access to any prior knowledge about the MDP model.

Learning by collecting experience in RL is accomplished via two different methods: model-based learning and model-free learning. Model-based RL attempts to first model the MDP structure (or its transition probabilities), and then based on the built model it synthesises the optimal policy via DP or other planning algorithms. The second method, model-free RL, learns an optimal policy directly by mapping state-action pairs to their expected reward, without the need for a model. Model-free RL is proved to converge to the same action selection policy as DP (over the original MDP) under mild assumptions [3, 12]. Both RL methods are extensively used in a variety of applications from robotics [13, 14], resource management [15, 16], traffic management [17] and flight control [18], chemistry [19], and gaming [20, 21].

Classical RL is focused on problems where the MDP states are finite. Nonetheless, many interesting real-world control tasks require actions to be taken in response to high-dimensional or real-valued sensory inputs [22]. As an example, consider the problem of drone control, in which the drone state is represented as its Euclidean position $(x, y, z) \in \mathbb{R}^3$: the physical space of an MDP modelling the stochastic behaviour of the drone is uncountably infinite, namely continuous.

The simplest way to solve an (uncountably) infinite-state MDP with RL is to discretise the state space and to resort to conventional RL to find the optimal policy [10]. Unfortunately, the resulting discrete model can be often inaccurate and may not capture the full dynamics of the original MDP, leading to a sub-optimal policy synthesis. One might argue that by increasing the number of discrete states this problem can be solved, however the more the discrete states, the more expensive and time-consuming the learning process is. Thus, MDP discretisation has to always deal with the trade off between accuracy and curse of dimensionality.

A more elaborate solution is to gather a set of experience samples and then use an approximation function constructed via regression from the samples set over the entire state space. A number of methods are available to approximate the expected reward function, e.g. sparse coarse coding [23], kernel-based modelling [24], tree-based regression [25], basis functions [26], etc. Among these methods, neural networks offer great promise in approximating the expected reward, due to their ability to generalise [27], and as a result there exist numerous successful applications of neural networks in RL for uncountably infinite or very large MDPs, e.g. TD-Gammon [28], Asynchronous Deep RL [29], Neural Fitted Q-iteration (NFQ) [30], CACLA [31] and Deep Q-networks (DQN) [32]. DQN are arguably one of the recent breakthroughs in RL, whereby human-level game play has been achieved on a number of Atari 2600 games. DQN attains this only by receiving available high-level information, namely the image visible on the game-screen and the score. This means that it is general enough that the rules of different games do not have to be explicitly encoded for the agents to learn successful control policies at the price of very high sample complexity.

Despite its generality, DQN is not a natural representation of how humans perceive these games, since humans already have prior knowledge and associations regarding many elements that appear on-screen and their corresponding function, e.g. "keys open doors". Given the useful domain knowledge that human experts can offer, and the otherwise huge challenge posed by randomly and exhaustively exploring large state spaces, new approaches have arisen that intend to combine human domain knowledge and insight with the ability of RL to eventually converge to near-optimal policies. These include apprenticeship learning, imitation learning, and expert demonstrations [33–41], and have already shown great improvements over the state-of-the-art learning methods. However, these approaches are very much biased towards the human behaviour and might not be able to find a global optimal control policy when the human teacher believes that a local optimum is actually global. Introducing useful

associations to RL in a formal way allows the agent to lift its initial knowledge about the problem and to efficiently find the global optimal policy, while avoiding an exhaustive exploration in the beginning or being biased towards human beliefs.

Contributions of This Work: Linear Temporal Logic (LTL) [42] is a formal language that offers a strong expressivity of formal engineering requirements and specifications, to the extent that there exists a substantial body of research on extraction of LTL properties from natural language, e.g. [43–45]. Given an LTL task, in this paper we propose the first framework for model-free RL algorithms, which allows to synthesise a control policy for an MDP such that the generated traces satisfy the LTL property with maximum probability. By employing LTL in an RL context, we can infuse structural knowledge into a learning procedure, whilst avoiding the bias otherwise introduced by a human teacher, as discussed above. This allows the expression of complex properties and extends cognate techniques, such as subtask decomposition and hierarchical learning. Among these properties, we can deal with safety, liveness and fairness guarantees. In particular, in order to show the enhancement of learning within the proposed architecture, we have picked the Atari 2600 game "Montezuma's Revenge" as one of our case studies, which is the only game in [32] that DQN fails to gain any score at.

Additionally, we prove that maximising the expected reward in our framework is equivalent to maximising the probability of satisfying the assigned LTL property, and we quantify this probability with a method based on asynchronous value iteration. Through theoretical results we show that whenever the probability of satisfying the given LTL property is zero, our algorithm produces "best available" policies. Another contribution of this work to handle time-varying periodic environments, which are given structure as Kripke transition systems that are then synchronised with the MDP.

In our setup, the LTL property acts as a high-level guide for the agent, whereas the low-level planning is handled by a native RL scheme. In order to synchronise this high-level guide with RL, we convert the LTL property into an automaton, namely a finite-state machine [46]. In general, however, LTL-to-automaton translation may result in a non-deterministic model, over which policy synthesis for MDPs is in general not semantically meaningful. A standard solution to this issue is to use Safra construction to determinise the automaton, which as expected can increase its size dramatically [47, 48]. An alternative solution is to directly convert the given LTL formula into a Deterministic Rabin Automaton (DRA), which by definition rules out non-determinism. Nevertheless, it is known that such conversion results, in the worst case, in automata that are doubly exponential in the size of the original LTL formula [49]. Conversely, in this paper we propose to express the given LTL property as a Limit Deterministic Büchi Automaton (LDBA) [50]. It is shown that this construction results in an exponential-sized automaton for

LTL\GU[1], and it results in nearly the same size as a DRA for the rest of LTL. Furthermore, a Büchi automaton is semantically easier than a Rabin automaton in terms of its acceptance conditions, which makes policy synthesis algorithms much simpler to implement [51].

Once the LDBA is generated from the given LTL property, we construct on-the-fly[2] a synchronous product between the MDP and the resulting LDBA and then define an adaptive reward function based on the acceptance condition of the Büchi automaton over the state-action pairs of the MDP. Using this algorithmic reward shaping procedure, RL is able to generate a policy (or policies) that returns the maximum expected reward, or as we will show, a policy (or policies) that satisfies the given LTL property with maximal probability. We also propose a mechanism to determine this probability while the agent is learning the MDP. Consequently, we can certify the generated policy by quantifying how safe it is with respect to the LTL property.

This work shows that the proposed architecture performs efficiently and is compatible with RL algorithms that are at the core of recent developments in the community, e.g. [29, 32]. Thus, we believe that the proposed approach can open up to further research in the area.

Related Work: The problem of control synthesis in finite-state MDPs with temporal logic has been considered in numerous works. In [52], the property of interest is expressed in LTL, which is converted to a DRA using standard methods. A product MDP is then constructed with the resulting DRA and a modified DP is applied over the product MDP, maximising the worst-case probability of satisfying the specification over all transition probabilities. However, [52] assumes to know the MDP a priori. [53] assumes that the given MDP model has unknown transition probabilities and builds a Probably Approximately Correct MDP (PAC MDP), which is multiplied by the logical property after conversion to DRA. The overall goal is to calculate the finite-horizon T-step value function for each state, such that the obtained value is within an error bound from the probability of satisfying the given LTL property. The PAC MDP is generated via an RL-like algorithm, then value iteration is applied to update state values.

The problem of policy generation by maximising the probability of satisfying given unbounded reachability properties is investigated in [54]. The policy generation relies on an approximate DP, even when the MDP transition probabilities are unknown. This requires a mechanism to approximate these probabilities (much like PAC MDP above), and the quality of the generated policy critically depends on the accuracy of this approximation. Therefore, a sufficiently large number of simulations has to be executed to make sure that the probability approximations are accurate enough [54]. Furthermore, the algorithm in [54] assumes prior knowledge about the smallest transition probability.

[1] LTL\GU is a fragment of linear temporal logic with the restriction that no until operator occurs in the scope of an always operator

[2] On-the-fly here means that the algorithm tracks (or executes) the state of an underlying structure (or a function) without explicitly building the entire structure a-priori.

Via LTL-to-DRA conversion, [54] algorithm can be extended to the problem of control synthesis for LTL specifications, at the expense double exponential blow-up of the obtained automaton. Much in the same direction, [17] employs a learning-based approach to generate a policy that is able to satisfy a given LTL property. For this approach, as remarked before, LTL-to-DRA conversion is in general known to result in large automata, and the reward shaping is complicated, due to the accepting conditions of the DRA. As for [54], the algorithm in [17] hinges on approximating the transition probabilities, which limits the precision of the policy generation process.

Compared to the mentioned approaches, the proposed framework learns the dynamics of the MDP implicitly, whilst synthesising the optimal policy at the same time, hence without explicitly having to construct the transition probabilities or the MDP model first. Indeed, the proposed framework can be implemented completely "model-free", which means that we are able to synthesise policies (1) without knowing MDP graph and its transition probabilities (as opposed to DP); and (2) without preprocessing or constructing a model of the MDP (which is the base for, among other techniques, model-based RL). The second feature results in the synthesis of policies by direct interaction with the MDP. Moreover, unlike [17], the proposed algorithms are able to find the optimal policy even if the satisfaction probability is not equal to one. In the RL literature, model-free methods are very successful, since they learn a direct mapping from states and actions to the associated expected reward. Alternative approaches, known as model-based learning, are not as general as model-free methods [55], even though they have convenient theoretical guarantees [56, 57].

Moving away from RL and full LTL, the problem of synthesising a policy that satisfies a temporal logic specification and that at the same time optimises a performance criterion is considered in [58–61]. In [61], the authors separate the problem into two sub-problems: extracting a (maximally) permissive strategy for the agent and then quantifying the performance criterion as a reward function and computing an optimal strategy for the agent within the operating envelope allowed by the permissive strategy. Similarly, [62] first computes safe, permissive strategies with respect to a reachability property. Then, under these constrained strategies, RL is applied to synthesise a policy that satisfies an expected cost criterion.

In [63], scLTL is proposed for mission specification, which results in deterministic finite automata. A product MDP is then constructed and a linear programming solver is used to find optimal policies. PCTL specifications are investigated in [64], where a linear optimisation solution is used to synthesise a control policy. In [65], an automated method is proposed to verify and repair the policies that are generated by RL with respect to a PCTL formula - the key engine runs by feeding the Markov chain induced by the policy to a probabilistic model checker. In [66], some practical challenges of RL are addressed by letting the agent plan ahead in real time using constrained optimisation.

Unfortunately, in the domain of continuous-state MDPs, to the best of our knowledge, no research has been done to enable RL to generate policies while respecting full LTL properties. The framework proposed in this paper

is the first algorithm that can handle both finite-state and continuous-state MDPs in this context. Probabilistic reachability over a finite horizon for hybrid continuous-state MDPs is investigated in [67], where a DP-based algorithm is employed to produce safe policies. DFAs have been employed in [68] to find an optimal policy for infinite-horizon probabilistic reachability problems. FAUST^2 [7] deals with uncountable-state MDPs by generating a discrete-state abstraction based on the knowledge of the MDP model. Using probabilistic bisimulation [69] showed that abstraction-based model checking can be effectively employed to generate control policies in continuous-state MDPs. Truncated LTL is proposed in [70] as the specification language, and a policy search method is used for synthesis. Focusing exclusively on the safety fragment of LTL, the concept of shielding is employed in [71] to synthesise a reactive system that ensures that the agent remains safe during and after learning. This approach is closely related to teacher-guided RL [72], since a shield can be considered as a teacher, which provides safe actions when absolutely necessary. To express the specification, [71] uses DFAs and then translates the problem into a safety game. The game is played by the environment and the agent. In every state of the game, the environment chooses an input, and then the agent selects an output. The game is won by the agent if only safe states are visited during the play. However, the generated policy always needs the shield to be online, as the shield maps every unsafe action to a safe action.

This article is organised as follow: Section 2 reviews basic concepts and definitions. In Section 3, we discuss the policy synthesis problem and we propose a method to constrain it. Case studies are provided in Section 4 to quantify the performance of the proposed algorithms. Extra material on this paper, including videos of the experiments, can be found at

https://www.cs.ox.ac.uk/conferences/lcrl

2 Background

2.1 Problem Setup

Definition 1 (Continuous-state Space MDP) The tuple $\mathbf{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \mathcal{AP}, L)$ is an MDP over a set of states $\mathcal{S} = \mathbb{R}^n$, where $\mathcal{A}$ is a finite set of actions, s_0 is the initial state and $P : \mathcal{B}(\mathbb{R}^n) \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a Borel-measurable transition kernel which assigns to any pair of state and action a probability measure on the Borel space $(\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n))$ [73]. $\mathcal{AP}$ is a finite set of atomic propositions and a labelling function $L : \mathcal{S} \to 2^{\mathcal{AP}}$ assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \subseteq 2^{\mathcal{AP}}$ [74].

A finite-state MDP is a special case of continuous-state MDP in which $|\mathcal{S}| < \infty$ and $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function. The transition function P induces a matrix, known as transition probability matrix.

Definition 2 (Path) In an MDP $\mathbf{M}$, an infinite path ρ starting at s_0 is a sequence of states $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ...$ such that every transition $s_i \xrightarrow{a_i} s_{i+1}$

is possible in $\mathbf{M}$, i.e. s_{i+1} belongs to the smallest Borel set B such that $P(B|s_i, a_i) = 1$ (or in a finite-state MDP, s_{i+1} is such that $P(s_{i+1}|s_i, a_i) > 0$). We might also denote ρ as $s_0..$ to emphasize that ρ starts from s_0.

Definition 3 (Stationary Policy) A stationary (randomized) policy $Pol : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a mapping from each state $s \in \mathcal{S}$, and action $a \in \mathcal{A}$ to the probability of taking action a in state s. A deterministic policy is a degenerate case of a randomized policy which outputs a single action at a given state, that is $\forall s \in \mathcal{S},\ \exists a \in \mathcal{A},\ Pol(s, a) = 1$.

In an MDP $\mathbf{M}$, we define a function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_0^+$ that denotes the immediate scalar bounded reward received by the agent from the environment after performing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.

Definition 4 (Expected Infinite-Horizon Discounted Reward) For a policy Pol on an MDP $\mathbf{M}$, and given a reward function R, the expected discounted reward at state s is defined as [11]:

$$U^{Pol}(s) = \mathbb{E}^{Pol}[\sum_{n=0}^{\infty} \gamma^n \ R(s_n, Pol(s_n))|s_0 = s], \tag{1}$$

where $\mathbb{E}^{Pol}[\cdot]$ denotes the expected value given that the agent follows policy Pol from state s, and $\gamma \in [0, 1)$ is a discount factor.

Definition 5 (Optimal Policy) An optimal policy Pol^* is defined as follows:

$$Pol^*(s) = \arg\sup_{Pol \in \mathcal{D}} U^{Pol}(s),$$

where $\mathcal{D}$ is the set of stationary deterministic policies over the state space $\mathcal{S}$.

Theorem 1 (From [1, 75]) *In any MDP $\boldsymbol{M}$ with a bounded reward function and a finite action space, if there exists an optimal policy, then that policy is stationary and deterministic.*

As discussed before, an MDP $\mathbf{M}$ is said to be solved if the agent discovers an optimal policy $Pol^* : \mathcal{S} \to \mathcal{A}$ that maximises the expected reward. Note that the reward function is assumed to be known in that we specify over which state-action pairs the agent will receive a given reward. Following this reward, the agent can generate an optimal policy in an unknown MDP.

In the following we provide the necessary background on model-free RL algorithms that we used in this work.

2.1.1 Finite-state MDPs

Q-learning (QL) is the most extensively used RL algorithm for synthesising optimal policies in finite-state MDPs [11]. For each state $s \in \mathcal{S}$ and for any available action $a \in \mathcal{A}$, QL assigns a quantitative value $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which is initialized with an arbitrary and finite value over all state-action pairs. As

the agent starts receiving rewards and learning, the Q-function is updated by the following rule when the agent takes action a at state s:

$$Q(s,a) \leftarrow Q(s,a) + \mu[R(s,a) + \gamma \max_{a' \in \mathcal{A}}(Q(s',a')) - Q(s,a)], \tag{2}$$

where $Q(s,a)$ is the Q-value corresponding to state-action (s,a), $0 < \mu \leq 1$ is called learning rate or step size, $R(s,a)$ is the reward obtained for performing action a in state s, $0 \leq \gamma \leq 1$ is the discount factor, and s' is the state obtained after performing action a. The Q-function for the rest of the state-action pairs remains unchanged.

Under mild assumptions over the learning rate, for finite-state and -action spaces QL converges to a unique limit[3], call it Q^*, as long as every state action pair is visited infinitely often [12]. Once QL converges, the optimal policy $Pol^* : \mathcal{S} \to \mathcal{A}$ can be generated by selecting the action that yields the highest Q^*, i.e.,

$$Pol^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \ Q^*(s,a).$$

Here Pol^* corresponds to the optimal policy that can be generated via DP. This means that when QL converges, we have

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s') U^{Pol^*}(s'), \tag{3}$$

where s' is the agent new state after choosing action a at state s such that $P(s'|s,a) > 0$.

2.1.2 Continuous-state MDPs

In QL the agent stores the Q-values possibly over all state-action pairs, and updates them according to the rule in (2). When the MDP has a continuous state space it is not possible to directly use standard QL. Thus, as mentioned earlier we have to either finitely discretise the state space or to turn to function approximations, in order to interpolate the Q-function over the entire uncountably infinite state space.

Neural Fitted Q-iteration (NFQ) [30] is an algorithm that employs neural networks [76] to approximate the Q-function, namely to efficiently generalise or interpolate it over the entire state space, exploiting a finite set of experience samples. NFQ is also the core engine behind the known DQN [32] architecture.

Instead of the update rule in (2), NFQ introduces a loss function that measures the error between the current Q-values $Q(s,a)$ and their target value $R(s,a) + \gamma \max_{a'} Q(s',a')$, namely

$$L = (Q(s,a) - R(s,a) + \gamma \max_{a'} Q(s',a'))^2. \tag{4}$$

[3] This unique limit is the expected discounted reward by taking action a at state s, and following the optimal policy afterwards.

Gradient descent techniques [77] are then applied to adjust the weights of the neural network, so that this loss is minimised.

In classical QL, the Q-function is updated whenever a state-action pair is visited. In the continuous state-space case, we may update the approximation likewise, i.e., update the neural net weights once a new state-action pair is visited. However, in practice, a large number of trainings might need to be carried out until an optimal or near-optimal policy is found. This is due to the uncontrollable variations occurring in the Q-function approximation caused by unpredictable changes in the network weights when the weights are adjusted for a specific state-action pair [78]. More precisely, if at each iteration we only introduce a single sample point the training algorithm tries to adjust the weights of the entire neural network, such that the loss function is minimised at that specific sample point. This might result in some changes in the network weights such that the error between the network output and the output of previous sample points becomes large and thus fails to approximate the Q-function correctly. Therefore, one needs to make sure that when the weights of the neural network are updated, we also consider all the previously generated samples: this technique is called "experience replay" [79], and is detailed next. The main idea underlying NFQ is to store all previous experiences and then reuse this data iteratively to update the neural Q-function. NFQ can thus be seen as a batch learning method in which there exists a training set that is repeatedly used to train the agent. In other words, experience gathering and learning happens separately.

NFQ exploits the positive effects of generalisation in neural nets as they are quite efficient in predicting Q-values for state-action pairs that have not been visited by interpolating between available data. This means that the learning algorithm requires less experience and the learning process is thus data efficient.

2.2 Linear Temporal Logic Properties

In the proposed architecture, we use LTL formulae to express a wide range of properties (e.g., temporal, sequential, conditional) and to systematically and automatically shape a corresponding reward: such reward would otherwise be hard (if at all possible) to express and achieve by conventional methods in classical reward shaping. LTL formulae over a given set of atomic propositions $\mathcal{AP}$ are syntactically defined as [42]

$$\varphi ::= true \mid \alpha \in \mathcal{AP} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi \cup \varphi, \tag{5}$$

where the operators $\bigcirc$ and $\cup$ are called "next" and "until", respectively. The semantics of LTL formulae, as interpreted over MDPs, are discussed in the following.

Given a path ρ, the i-th state of ρ is denoted by $\rho[i]$, where $\rho[i] = s_i$. Furthermore, the i-th suffix of ρ is $\rho[i..]$ where $\rho[i..] = s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} s_{i+2} \xrightarrow{a_{i+2}} s_{i+3} \xrightarrow{a_{i+3}} \dots$.

Definition 6 (LTL Semantics) For an LTL formula φ and for a path ρ, the satisfaction relation $\rho \models \varphi$ is defined as

$\rho \models \alpha \in \mathcal{AP} \Leftrightarrow \alpha \in L(\rho[0])$,
$\rho \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \rho \models \varphi_1 \wedge \rho \models \varphi_2$,
$\rho \models \neg\varphi \Leftrightarrow \rho \not\models \varphi$,
$\rho \models \bigcirc\varphi \Leftrightarrow \rho[1..] \models \varphi$,
$\rho \models \varphi_1 \cup \varphi_2 \Leftrightarrow \exists j \in \mathbb{N}_0$ s.t. $\rho[j..] \models \varphi_2$ *and* $\forall i,\ 0 \leq i < j,\ \rho[i..] \models \varphi_1$.

Through the until operator we are furthermore able to define two temporal modalities: (1) eventually, $\Diamond\varphi = true \cup \varphi$; and (2) always, $\Box\varphi = \neg\Diamond\neg\varphi$. An LTL formula φ over $\mathcal{AP}$ specifies the following set of words:

$$Words(\varphi) = \{\sigma \in (2^{\mathcal{AP}})^\omega \text{ s.t. } \sigma \models \varphi\}. \tag{6}$$

Definition 7 (Probability of Satisfying an LTL Formula) Starting from any state s, we denote the probability of satisfying formula φ as

$$Pr(s..^{Pol} \models \varphi),$$

where $s..^{Pol}$ is the collection of all paths starting from s, generated under policy Pol.

Definition 8 (Policy Satisfaction) In an MDP $\mathbf{M}$, we say that a stationary deterministic policy Pol satisfies an LTL formula φ if:

$$Pr(s_0..^{Pol} \models \varphi) > 0,$$

where s_0 is the initial state of the MDP.

Using an LTL formula we can now specify a set of constraints (i.e., requirements, or specifications) over the traces of the MDP. Once a policy Pol is selected, it dictates which action has to be taken at each state of the MDP $\mathbf{M}$. Hence, the MDP $\mathbf{M}$ is reduced to a Markov chain, which we denote by $\mathbf{M}^{Pol}$.

For an LTL formula φ, an alternative method to express the set $Words(\varphi)$ in (6) is to employ a limit-deterministic Büchi automaton (LDBA) [50]. We first define a Generalized Büchi Automaton (GBA), then we formally introduce the LDBA [50].

Definition 9 (Generalized Büchi Automaton) A GBA $\mathbf{N} = (\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ is a structure where $\mathcal{Q}$ is a finite set of states, $q_0 \in \mathcal{Q}$ is the initial state, $\Sigma = 2^{\mathcal{AP}}$ is a finite alphabet, $\mathcal{F} = \{F_1, ..., F_f\}$ is the set of accepting conditions, where $F_j \subset \mathcal{Q}, 1 \leq j \leq f$, and $\Delta : \mathcal{Q} \times \Sigma \to 2^{\mathcal{Q}}$ is a transition relation.

Let Σ^ω be the set of all infinite words over Σ. An infinite word $w \in \Sigma^\omega$ is accepted by a GBA $\mathbf{N}$ if there exists an infinite run $\theta \in \mathcal{Q}^\omega$ starting from q_0 where $\theta[i+1] \in \Delta(\theta[i], \omega[i]),\ i \geq 0$ and for each $F_j \in \mathcal{F}$

$$inf(\theta) \cap F_j \neq \emptyset, \tag{7}$$

where $inf(\theta)$ is the set of states that are visited infinitely often by the run θ.

Definition 10 (LDBA) A GBA $\mathbf{N} = (\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ is limit-deterministic if $\mathcal{Q}$ can be partitioned into two disjoint sets $\mathcal{Q} = \mathcal{Q}_N \cup \mathcal{Q}_D$, such that [50]:

- $\Delta(q, \alpha) \subset \mathcal{Q}_D$ and $|\Delta(q, \alpha)| = 1$ for every state $q \in \mathcal{Q}_D$ and for every $\alpha \in \Sigma$,
- for every $F_j \in \mathcal{F}$, $F_j \subset \mathcal{Q}_D$.

Intuitively, an LDBA is a GBA that has two partitions: initial ($\mathcal{Q}_N$) and accepting ($\mathcal{Q}_D$). The accepting part includes all the accepting states and has deterministic transitions.

Remark 1 The LTL-to-LDBA algorithm proposed in [50], which is used in this paper, results in an automaton with two parts (initial $\mathcal{Q}_N$ and accepting $\mathcal{Q}_D$). Both initial and accepting parts comprise deterministic transitions, and additionally there are non-deterministic ε-transitions between them. According to Definition 10, the discussed structure is still an LDBA (the determinism in the initial part is stronger than that required in the LDBA definition). An ε-transition allows an automaton to change its state without reading an input symbol. In practice, during the implementation of RL, the ε-transitions between $\mathcal{Q}_N$ and $\mathcal{Q}_D$ reflect the agent's "guess" on reaching $\mathcal{Q}_D$: accordingly, if after an ε-transition the associated labels in the accepting set of the automaton cannot be read, or the accepting states cannot be visited, then the guess is deemed to be wrong, and the trace is disregarded. ⌟

3 Logically-Constrained Reinforcement Learning (LCRL)

We are interested in synthesising a policy (or policies) for an unknown MDP via RL such that the obtained structure satisfies a given LTL property. In order to explain the core ideas of the algorithm and for ease of exposition, we assume that the MDP graph and the associated transition probabilities are known. Later these assumptions are entirely removed, and we stress that the algorithm can be run model-free. We relate the MDP and the automaton by synchronising them, in order to create a new structure that is first of all compatible with RL and secondly that encompasses the given logical property.

Definition 11 (Product MDP) Given an MDP $\mathbf{M}(\mathcal{S}, \mathcal{A}, s_0, P, \mathcal{AP}, L)$ and an LDBA $\mathbf{N}(\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ with $\Sigma = 2^{\mathcal{AP}}$, the product MDP is defined as $(\mathbf{M} \otimes \mathbf{N}) = \mathbf{M}_{\mathbf{N}}(\mathcal{S}^{\otimes}, \mathcal{A}, s_0^{\otimes}, P^{\otimes}, \mathcal{AP}^{\otimes}, L^{\otimes}, \mathcal{F}^{\otimes})$, where $\mathcal{S}^{\otimes} = \mathcal{S} \times \mathcal{Q}$, $s_0^{\otimes} = (s_0, q_0)$, $\mathcal{AP}^{\otimes} = \mathcal{Q}$, $L^{\otimes} : \mathcal{S}^{\otimes} \to 2^{\mathcal{Q}}$ such that $L^{\otimes}(s, q) = q$ and $\mathcal{F}^{\otimes} \subseteq \mathcal{S}^{\otimes}$ is the set of accepting states $\mathcal{F}^{\otimes} = \{F_1^{\otimes}, ..., F_f^{\otimes}\}$, where $F_j^{\otimes} = \mathcal{S} \times F_j$. The transition kernel $P^{\otimes}$ is such that given the current state (s_i, q_i) and action a, the new state is (s_j, q_j), where $s_j \sim P(\cdot|s_i, a)$ and $q_j \in \Delta(q_i, L(s_j))$. When the MDP $\mathbf{M}$ has a finite state space, then $P^{\otimes} : \mathcal{S}^{\otimes} \times \mathcal{A} \times \mathcal{S}^{\otimes} \to [0, 1]$ is the transition probability function, such that $(s_i \xrightarrow{a} s_j) \wedge (q_i \xrightarrow{L(s_j)} q_j) \Rightarrow P^{\otimes}((s_i, q_i), a, (s_j, q_j)) = P(s_i, a, s_j)$. Furthermore, in order to handle ε-transitions we make the following modifications to the above definition of product MDP:

– for every potential ε-transition to some state $q \in \mathcal{Q}$ we add a corresponding action ε_q in the product:

$$\mathcal{A}^{\otimes} = \mathcal{A} \cup \{\varepsilon_q, q \in \mathcal{Q}\}.$$

– The transition probabilities corresponding to ε-transitions are given by

$$P^{\otimes}((s_i, q_i), \varepsilon_q, (s_j, q_j)) = \begin{cases} 1 & \text{if } s_i = s_j,\ q_i \xrightarrow{\varepsilon_q} q_j = q, \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, by constructing the product MDP we add an extra dimension to the state space of the original MDP. The role of the added dimension is to track automaton states and, hence, to synchronise the current state of the MDP with the state of the automaton: this allows to evaluate the satisfaction of the corresponding LTL property (or parts thereof). Before introducing a reward assignment for RL, we need to define the ensuing notions.

Definition 12 (Non-accepting Sink Component) A non-accepting sink component of the LDBA $\mathbf{N} = (\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ is a directed graph induced by a set of states $Q \subset \mathcal{Q}$ such that (1) the graph is strongly connected; (2) it does not include all accepting sets $F_k,\ k = 1, ..., f$; and (3) there exist no other set $Q' \subset \mathcal{Q},\ Q' \neq Q$ that $Q \subset Q'$. We denote the union of all non-accepting sink components as *SINKs*. Note that *SINKs* can be an empty set.

Definition 13 (Accepting Frontier Function) For an LDBA $\mathbf{N} = (\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$, we define the function $Acc : \mathcal{Q} \times 2^{\mathcal{Q}} \to 2^{\mathcal{Q}}$ as the accepting frontier function, which executes the following operation over a given set $\mathbb{F} \in 2^{\mathcal{Q}}$

$$Acc(q, \mathbb{F}) = \begin{cases} \mathbb{F} \setminus F_j & q \in F_j \wedge \mathbb{F} \neq F_j, \\ \bigcup_{k=1}^{f} F_k \setminus F_j & q \in F_j \wedge \mathbb{F} = F_j, \\ \mathbb{F} & \text{otherwise.} \end{cases}$$

Once the state $q \in F_j$ and the set $\mathbb{F}$ are introduced to the function *Acc*, it outputs a set containing the elements of $\mathbb{F}$ minus those elements that are common with F_j. However, if $\mathbb{F} = F_j$, then the output is the union of all accepting sets of the LDBA minus those elements that are common with F_j. Finally, if the state q is not an accepting state then the output of *Acc* is $\mathbb{F}$.

Remark 2 In order to clearly elucidate the role of different components in the proposed approach, we have employed model-dependent notions, such as transition probabilities and the product MDP. However, we emphasise that the proposed approach can run "model-free", and as such it does not depend on these components. In particular, as per Definition 10, the LDBA is composed of two disjoint sets of states $\mathcal{Q}_D$ (which is invariant) and $\mathcal{Q}_N$, where the accepting states belong to the set $\mathcal{Q}_D$. Since all transitions are deterministic within $\mathcal{Q}_N$

and $\mathcal{Q}_D$, the automaton transitions can be executed "only" by reading the labels, which makes the agent aware of the automaton state without explicitly constructing the product MDP. We will later define a reward function "on-the-fly", emphasising that the agent does not need to know the model structure or the transition probabilities (or their product). ⌟

The product MDP encompasses transition relations of the original MDP and the structure of the Büchi automaton, and it inherits characteristics of both. A proper reward function leads the RL agent to find a policy that is optimal, in the sense that it satisfies the LTL property φ with maximal probability. We employ an on-the-fly reward function that fits the RL architecture: when an agent observes the current state $s^{\otimes}$, implements action a and observes the subsequent state $s^{\otimes'}$, the reward provides the agent with a scalar value, according to the following reward:

$$R(s^{\otimes}, a) = \begin{cases} r_p & \text{if } q' \in \mathbb{A},\ s^{\otimes'} = (s', q'), \\ r_n & \text{otherwise.} \end{cases} \tag{8}$$

Here $r_p = M + y \times m \times rand(s^{\otimes})$ is a positive reward and $r_n = y \times m \times rand(s^{\otimes})$ is a neutral reward. The parameter $y \in \{0, 1\}$ is a constant, $0 < m \ll M$ are arbitrary positive values, and $rand : \mathcal{S}^{\otimes} \to (0, 1)$ is a function that generates a random number in $(0, 1)$ for each state $s^{\otimes}$ each time R is being evaluated. The role of the function *rand* is to break the symmetry when neural nets are used for approximating the Q-function[4]. Also, note that parameter y acts as a switch to bypass the effect of the *rand* function on R. As we will see shortly, this switch is active in a few algorithms, and disabled in others.

The set $\mathbb{A}$ is called the accepting frontier set, is initialised as $\mathbb{A} = \bigcup_{k=1}^{f} F_k$, and is updated by the following rule every time after the reward function is evaluated:

$$\mathbb{A} \leftarrow Acc(q', \mathbb{A}).$$

The set $\mathbb{A}$ always contains those accepting states that are needed to be visited at a given time: in this sense the reward function is "adaptive" to the accepting condition set by the LDBA. Thus, the agent is guided by the above reward assignment to visit these states and once all of the sets $F_k,\ k = 1, ..., f$, are visited, the accepting frontier $\mathbb{A}$ is reset. As such, the agent is guided to visit the accepting sets infinitely often, and consequently, to satisfy the given LTL property.

Remark 3 Note that when running our algorithm there is no need to "explicitly build" the product MDP and to store all its states in memory. The automaton

[4] If all weights in a feedforward net start with equal values and if the solution requires that unequal weights be developed, the network can never learn. The reason is that the correlations between the weights within the same hidden layer can be described by symmetries in that layer, i.e. identical weights. Therefore, the neural net can generalise if such symmetries are broken and the redundancies of the weights are reduced. Starting with completely identical weights prevents the neural net from minimising these redundancies and from optimising the loss function [27].

transitions can be executed on-the-fly as the agent reads the labels of the MDP states. ⌟

In the following, we evaluate the proposed architecture in both finite- and continuous-state MDPs.

3.1 Finite-state MDPs: Logically-Constrained QL

Over finite-state MDPs, as introduced above we run QL over the product MDP $\mathbf{M_N}$ with the reward shaping proposed in (8), where we have set $y = 0$. In order to handle also non-ergodic MDPs, we propose to employ a variant of standard QL that consists of several resets, at each of which the agent is forced to re-start from its initial state s_0. Each reset defines an episode, as such the algorithm is called "episodic QL". However, for the sake of simplicity, we omit the term "episodic" in the rest of the paper and we use the term Logically-Constrained QL (LCQL).

As stated earlier, since QL is proved to converge to the optimal Q-function, we can synthesise the optimal policy in the limit. The following result shows that the optimal policy produced by LCQL indeed satisfies the given LTL property (Definition 8).

Theorem 2 *Let the MDP $\boldsymbol{M_N}$ be the product of an MDP $\boldsymbol{M}$ and an LDBA $\boldsymbol{N}$ that is associated with the given LTL property φ. If a satisfying policy exists, then the LCQL algorithm will in the limit find one such policy.*

Proof Assume that there exists a policy $\overline{Pol}$ that satisfies φ. Policy $\overline{Pol}$ induces a Markov chain $\mathbf{M_N^{\overline{Pol}}}$ when it is applied over the MDP $\mathbf{M_N}$. This Markov chain comprises a disjoint union between a set of transient states $T_{\overline{Pol}}$ and h sets of irreducible recurrent classes $R^i_{\overline{Pol}},\ i = 1, ..., h$ [74], namely:

$$\mathbf{M_N^{\overline{Pol}}} = T_{\overline{Pol}} \sqcup R^1_{\overline{Pol}} \sqcup ... \sqcup R^h_{\overline{Pol}}.$$

From (7), policy $\overline{Pol}$ satisfies φ if and only if:

$$\exists R^i_{\overline{Pol}} \text{ s.t. } \forall j \in \{1, ..., f\},\ F_j^{\otimes} \cap R^i_{\overline{Pol}} \neq \emptyset. \tag{9}$$

The recurrent classes that satisfy (9) are called accepting. From the irreducibility of the recurrent class $R^i_{\overline{Pol}}$ we know that all the states in $R^i_{\overline{Pol}}$ communicate with each other thus, once a trace ends up in such set, then all the accepting sets are going to be visited infinitely often. Therefore, from the definition of $\mathbb{A}$ and of the accepting frontier function (Definition 13), the agent receives a positive reward r_p ever after it has reached an accepting recurrent class $R^i_{\overline{Pol}}$.

There are two other possibilities concerning the remaining recurrent classes that are not accepting. A non-accepting recurrent class, name it $R^k_{\overline{Pol}}$, either

1. has no intersection with any accepting set $F_j^{\otimes}$, i.e.

$$\forall j \in \{1, ..., f\},\ F_j^{\otimes} \cap R^k_{\overline{Pol}} = \emptyset;$$

2. or has intersection with some of the accepting sets but not all of them, i.e.

$$\exists J \subset 2^{\{1,...,f\}} \setminus \{1, ..., f\} \text{ s.t. } \forall j \in J,\ F_j^{\otimes} \cap R^k_{\overline{Pol}} \neq \emptyset.$$

In the first instance, the agent does not visit any accepting set in the recurrent class and the likelihood of visiting accepting sets within the transient states $T_{\overline{Pol}}$ is zero since $\mathcal{Q}_D$ is invariant.

In the second case, the agent is able to visit some accepting sets but not all of them. This means that in the update rule of the frontier accepting set $\mathbb{A}$ in Definition 13, the case where $q \in F_j \wedge \mathbb{A} = F_j$ will never happen since there exist always at least one accepting set that has no intersection with $R^k_{\overline{Pol}}$. Therefore, after a limited number of times, no positive reward can be obtained, and the reinitialisation of $\mathbb{A}$ in Definition 13 is blocked.

Recall Definition 4, where the expected reward for the initial state $\bar{s} \in \mathcal{S}^{\otimes}$ is defined as:

$$U^{\overline{Pol}}(\bar{s}) = \mathbb{E}^{\overline{Pol}}[\sum_{n=0}^{\infty} \gamma^n\ R(S_n, \overline{Pol}(S_n))|S_0 = \bar{s}].$$

In both cases, for any arbitrary $r_p > 0$ (and $r_n = 0$), there always exists a γ such that the expected reward of a trace hitting $R^i_{\overline{Pol}}$ with unlimited number of positive rewards, is higher than the expected reward of any other trace.

In the following, by contradiction, we show that any optimal policy Pol^* which optimises the expected reward will satisfy the property. Suppose then that the optimal policy Pol^* does not satisfy the property φ. By this assumption:

$$\forall R^i_{Pol^*},\ \exists j \in \{1, ..., f\},\ F_j^{\otimes} \cap R^i_{Pol^*} = \emptyset. \tag{10}$$

As we discussed in case 1 and case 2 above, the accepting policy $\overline{Pol}$ has a higher expected reward than the optimal policy Pol^* due to limited number of positive rewards in policy Pol^*. This is, however, in direct contrast with Definition 5, leading to a contradiction. ⌟

Remark 4 Note that LCQL outputs its policy by choosing the maximum Q-value at any given state. Further, as we will show in Theorem 3, we can derive the probability of satisfying the LTL property from the Q-values. This means that, if there exists more than one optimal policy, i.e. if there is more than one satisfying policy corresponding to the same probability, then LCQL is able to find all of them by presenting the same Q-values to the agent for these policies. Thus, the agent is free to choose between these policies by arbitrarily choosing actions that have the same expected reward. ⌟

Definition 14 (Closeness to Satisfaction) Assume that two policies Pol_1 and Pol_2 do not satisfy the property φ. Accordingly, there are accepting sets in the automaton that have no intersection with runs of induced Markov chains $\mathbf{M}^{Pol_1}$ and $\mathbf{M}^{Pol_2}$. We say that Pol_1 is closer to satisfying the property if runs of $\mathbf{M}^{Pol_1}$ have more intersections with accepting sets of the automaton than runs of $\mathbf{M}^{Pol_2}$.

Corollary 1 *If no policy in the MDP $\boldsymbol{M}$ can be generated to satisfy the property φ, LCQL yields in the limit the policy that is closest to satisfying the given LTL formula φ.*

Proof Assume that there exists no policy in MDP $\mathbf{M}$ that can satisfy the property φ. Construct the induced Markov chain $\mathbf{M}_{\mathbf{N}}^{Pol}$ for any arbitrary policy Pol and its associated set of transient states T_{Pol} and h sets of irreducible recurrent classes R_{Pol}^i: $\mathbf{M}_{\mathbf{N}}^{Pol} = T_{Pol} \sqcup R_{Pol}^1 \sqcup ... \sqcup R_{Pol}^h$. By assumption, policy Pol cannot satisfy the property and we thus have that $\forall R_{Pol}^i,\ \exists j \in \{1, ..., f\},\ F_j^{\otimes} \cap R_{Pol}^i = \emptyset$, which means that there are some automaton accepting sets like F_j that cannot be visited. Therefore, after a limited number of times no positive reward is given by the reward function $R(s^{\otimes}, a)$. However, the closest recurrent class to satisfying the property is the one that intersects with more accepting sets.

By Definition 4, for any arbitrary $r_p > 0$ (and $r_n = 0$), the expected reward at the initial state for a trace with highest number of intersections with accepting sets is maximum among other traces. Hence, by the convergence guarantees of QL, the optimal policy produced by LCQL converges to a policy whose recurrent classes of its induced Markov chain have the highest number of intersections with the accepting sets of the automaton. ⌟

Theorem 3 *If the LTL property is satisfiable by the MDP $\boldsymbol{M}$, then the optimal policy, maximising the expected reward, and generated by LCQL, maximises the probability of property satisfaction.*

Proof Assume that MDP $\mathbf{M_N}$ is the product of an MDP $\mathbf{M}$ and an automaton $\mathbf{N}$ where $\mathbf{N}$ is the automaton associated with the given LTL property φ. In MDP $\mathbf{M_N}$, a directed graph induced by a pair $(S^{\otimes}, A),\ S^{\otimes} \subseteq \mathcal{S}^{\otimes},\ A \subseteq \mathcal{A}$ is a Maximal End Component (MEC) if it is strongly connected and there exists no $(S^{\otimes\prime}, A')$ such that $(S^{\otimes}, A) \neq (S^{\otimes\prime}, A')$ and $S^{\otimes} \subset S^{\otimes\prime}$ and $A \subset A'$ for all $s^{\otimes} \in S^{\otimes}$ [46]. A MEC is accepting if it contains accepting conditions of the automaton associated with the property φ. The set of all accepting MECs are denoted by *AMECs*.

We first review how this probability is calculated traditionally when the MDP is fully known and then we show that LCQL convergence is the same. Normally when the MDP graph and transition probabilities are known, the probability of property satisfaction is often calculated via DP-based methods such as standard value iteration over the product MDP $\mathbf{M_N}$ [46]. This allows to convert the satisfaction problem into a reachability problem. The goal in this reachability problem is to find the maximum (or minimum) probability of reaching *AMECs*.

The value function $V : \mathcal{S}^{\otimes} \rightarrow [0,1]$ in value iteration is then initialised to 0 for non-accepting MECs and to 1 for the rest of the MDP. Once value iteration converges then at any given state $s^{\otimes} = (s,q) \in \mathcal{S}^{\otimes}$ the optimal policy $\pi^* : \mathcal{S}^{\otimes} \rightarrow \mathcal{A}$ is produced by

$$\pi^*(s^{\otimes}) = \arg\max_{a} \sum_{s^{\otimes\prime} \in \mathcal{S}} P(s^{\otimes}, a, s^{\otimes\prime}) V^*(s^{\otimes\prime}), \tag{11}$$

where V^* is the converged value function, representing the maximum probability of satisfying the property at state s. In the following we show that the optimal policy Pol^*, generated by LCQL, is indeed equivalent to π^*.

They key to compare standard model-checking methods to LCQL is reduction of value iteration to basic form. More specifically, quantitative model-checking over an MDP with a reachability predicate can be converted to a model-checking problem with an equivalent reward predicate which is called the basic form. This reduction is done by adding a one-off reward of 1 upon reaching *AMECs* [80]. Once this reduction is done, Bellman operation is applied over the value function (which represents the satisfaction probability) and policy π^* maximises the probability of satisfying the property.

In LCQL, when an AMEC is reached, all of the automaton accepting sets will surely be visited by policy Pol^* and an infinite number of positive rewards r_p will be given to the agent as shown in Theorem 2.

There are two natural ways to define the total discounted rewards [81]:

1. to interpret discounting as the coefficient in front of the reward.
2. to define the total discounted rewards as a terminal reward after which no reward is given and treat the update rule as if it is undiscounted.

It is well-known that the expected total discounted rewards corresponding to these methods are the same, e.g. [81]. Therefore, without loss of generality, given any discount factor γ, and any positive reward component r_p, the expected discounted reward for the discounted case (LCQL) is c times the undiscounted case (value iteration) where c is a positive constant. This concludes maximising one is equivalent to maximising the other. ⌟

So far we have discussed an RL implementation that is capable of synthesising policies that can respect an LTL formula over a finite-state MDP. In the following, we present an additional component, which allows to quantify the quality of the resulting policy by calculating the probability of satisfaction associated to the policy.

3.1.1 Probability of Satisfaction of a Property

The Probability of Satisfaction of a Property (PSP) can be calculated via standard DP, as implemented for instance in PRISM [82] and Storm [83]. However, as discussed before, DP is quite limited when the state space of the given MDP is large.

In this section we propose a local value iteration method as part of LCQL that calculates this probability in parallel with the RL scheme. RL guides the local update of the value iteration, such that it only focuses on parts of the state space that are relevant to the satisfaction of the property. This allows the value iteration to avoid an exhaustive search and thus to converge faster.

Recall that the transition probability function $P^{\otimes}$ is not known. Further, according to Definition 11, $P^{\otimes}((s_i, q_i), a, (s_j, q_j)) = P(s_i, a, s_j)$ if $(s_i \xrightarrow{a} s_j)$ and $(q_i \xrightarrow{L(s_j)} q_j)$, showing the intrinsic dependence of $P^{\otimes}$ on P. This allows to apply the definition of α-approximation in MDPs [84] as follows.

Let us introduce two functions $\Psi : \mathcal{S} \times \mathcal{A} \to \mathbb{N}$ and $\psi : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{N}$ over the MDP $\mathbf{M}$. Function $\psi(s, a, s')$ represents the number of times the agent executes action a in state s, thereafter moving to state s', whereas $\Psi(s, a) = \sum_{s' \in \mathcal{S}} \psi(s, a, s')$. The maximum likelihood of $P(s, a, s')$ is a Gaussian normal distribution with the mean $\bar{P}(s, a, s') = \psi(s, a, s')/\Psi(s, a)$, so that the variance of this distribution asymptotically converges to zero and $\bar{P}(s, a, s') = P(s, a, s')$. Function $\Psi(s, a)$ is initialised to be equal to one for every state-action pair (reflecting the fact that at any given state it is possible to take any action, and also avoiding division by zero), and function $\psi(s, a, s')$ is initialised to be equal to zero.

Consider a function $PSP : \mathcal{S}^{\otimes} \to [0, 1]$. For a given state $s^{\otimes} = (s, q)$, the PSP function is initialised as $PSP(s^{\otimes}) = 0$ if q belongs to *SINKs*. Otherwise, it is initialised as $PSP(s^{\otimes}) = 1$. Recall that *SINKs* are those components in the automaton that are surely non-acceptable and impossible to escape from.

Definition 15 (Optimal PSP) The optimal PSP vector is denoted by $\underline{PSP}^* = (PSP^*(s_1), ..., PSP^*(s_{|\mathcal{S}|}))$, where $PSP^* : \mathcal{S}^{\otimes} \to [0, 1]$ is the optimal PSP function and $PSP^*(s_i)$ is the optimal PSP value starting from state s_i such that

$$PSP^*(s_i) = \sup_{Pol \in \mathcal{D}} PSP^{Pol}(s_i),$$

where $PSP^{Pol}(s_i)$ is the PSP value of state s_i if we use the policy Pol to determine subsequent states.

In the following we prove that a proposed update rule that makes PSP converge to PSP^*.

Definition 16 (Bellman operation [3]) For any vector such as $\underline{PSP} = (PSP(s_1), ..., PSP(s_{|\mathcal{S}|}))$ in the MDP $\mathbf{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \mathcal{AP}, L)$, the Bellman DP operation T over the elements of $\underline{PSP}$ is defined as:

$$T\ PSP(s_i) = \max_{a \in \mathcal{A}_{s_i}} \sum_{s' \in \mathcal{S}} P(s_i, a, s') PSP(s'). \tag{12}$$

If the operation T is applied over all the elements of $\underline{PSP}$, we denote it as $T\ \underline{PSP}$.

Proposition 1 (From [3]) *The optimal PSP vector* $\underline{PSP}^*$ *satisfies the following equation:*

$$\underline{PSP}^* = T\ \underline{PSP}^*,$$

and additionally, $\underline{PSP}^*$ *is the "only" solution of the equation* $\underline{PSP} = T\ \underline{PSP}$.

In the standard value iteration method the value estimation is simultaneously updated for all states. However, an alternative method is to update the value for one state at a time. This method is known as asynchronous value iteration.

Definition 17 (Gauss-Seidel Asynchronous Value Iteration (AVI) [3]) We denote AVI operation by F and is defined as follows:

$$F\ PSP(s_1) = \max_{a\in\mathcal{A}}\{\sum_{s'\in\mathcal{S}} P(s_1,a,s')PSP(s')\}, \tag{13}$$

Algorithm 1: Logically-Constrained QL

input : LTL specification, *it_threshold*, γ, μ
output : Pol^* and $\underline{PSP}^*$
initialize $Q : \mathcal{S}^\otimes \times \mathcal{A}^\otimes \to \mathbb{R}_0^+$
initialize $PSP : \mathcal{S}^\otimes \to [0,1]$
initialize $\psi : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{N}$
initialize $\Psi : \mathcal{S} \times \mathcal{A} \to \mathbb{N}$
convert the desired LTL property to an LDBA $\mathbf{N}$
initialize $\mathbb{A}$
initialize *episode-number* $:= 0$
initialize *iteration-number* $:= 0$
while *Q is not converged* **do**
  episode-number $++$
  $s^\otimes = (s_0, q_0)$
  while $(q \notin SINKs :\ s^\otimes = (s,q))$ & (*iteration-number* $<$ *it_threshold*) **do**
    iteration-number $++$
    choose $a_* = Pol(s^\otimes) = \arg\max_{a\in\mathcal{A}} Q(s^\otimes, a)$ # or $\epsilon-$greedy with diminishing ϵ
    $\Psi(s^\otimes, a_*) ++$
    move to $s_*^\otimes$ by a_*
    if $\Psi(s^\otimes, a_*) = 2$ **then**
      $\psi(s^\otimes, a_*, s_*^\otimes) = 2$
    else
      $\psi(s^\otimes, a_*, s_*^\otimes) ++$
    end
    receive the reward $R(s^\otimes, a_*)$
    $\mathbb{A} \leftarrow Acc(s_*, \mathbb{A})$
    $Q(s^\otimes, a_*) \leftarrow Q(s^\otimes, a_*) + \mu[R(s^\otimes, a_*) - Q(s^\otimes, a_*) + \gamma \max_{a'}(Q(s_*^\otimes, a'))]$
    $\bar{P}^\otimes(s_i^\otimes, a, s_j^\otimes) \leftarrow \psi(s_i^\otimes, a, s_j^\otimes)/\Psi(s_i^\otimes, a),\ \forall s_i^\otimes, s_j^\otimes \in \mathcal{S}^\otimes$ and $\forall a \in \mathcal{A}$
    $PSP(s^\otimes) \leftarrow \max_{a\in\mathcal{A}} \sum_{s^{\otimes\prime}\in\mathcal{S}^\otimes} \bar{P}^\otimes(s^\otimes, a, s^{\otimes\prime}) \times PSP(s^{\otimes\prime})$
    $s^\otimes = s_*^\otimes$
  end
end

where s_1 is the state that current state at the MDP, and for all $s_i \neq s_1$:

$$\begin{aligned} F\ PSP(s_i) = \max_{a \in A} \{ & \sum_{s' \in \{s_1, \ldots, s_{i-1}\}} P(s_i, a, s') \times \\ F\ PSP(s') + & \sum_{s' \in \{s_i, \ldots, s_{|S|}\}} P(s_i, a, s') PSP(s') \}. \end{aligned} \tag{14}$$

By (14) we update the value of *PSP* state by state and use the calculated value for the next step.

Proposition 2 (From [3]) *Let $k_0, k_1, \ldots$ be an increasing sequence of iteration indices such that $k_0 = 0$ and each state is updated at least once between iterations k_m and $k_{m+1} - 1$, for all $m = 0, 1, \ldots$. Then the sequence of value vectors generated by AVI asymptotically converges to $\underline{PSP}^*$.*

Lemma 1 (From [3]) *F is a contraction mapping with respect to the infinity norm. In other words, for any two value vectors PSP and PSP′:*

$$||F\ \underline{PSP} - F\ \underline{PSP}'||_\infty \leq ||\underline{PSP} - \underline{PSP}'||_\infty.$$

Proposition 3 (Convergence, [84]) *From Lemma 1, and under the assumptions of Proposition 2, $\bar{P}(s, a, s')$ converges to $P(s, a, s')$, and from Proposition 2, the AVI value vector $\underline{PSP}$ asymptotically converges to $\underline{PSP}^*$, i.e. the probability that could be alternatively calculated by DP-based methods if the MDP was fully known.*

We conclude this section by presenting the overall procedure in Algorithm 1. The input of LCQL includes *it_threshold*, which is an upper bound on the number of iterations.

3.2 Continuous-state MDPs: Logically-Constrained NFQ

In this section, we propose the first RL algorithm based on Neural Fitted Q-iteration (NFQ) that can synthesise a policy satisfying a given LTL property when the given MDP has a continuous state space. We call this algorithm Logically-Constrained NFQ (LCNFQ).

In LCNFQ, in order to use the experience replay technique mentioned before, we let the agent explore the MDP and reinitialise it when a positive reward is received or when no positive reward is received after a given number *th* of iterations. The parameter *th* is set manually according to the state space of the MDP, allowing the agent to explore the MDP while keeping the size of the sample set limited. All episode traces, i.e. experiences, are stored in the form of $(s^\otimes, a, s^{\otimes\prime}, R(s^\otimes, a), q)$. Here $s^\otimes = (s, q)$ is the current state in the product MDP, a is the chosen action, $s^{\otimes\prime} = (s', q')$ is the resulting state, and $R(s^\otimes, a)$ is the obtained reward. The set of past experiences is called the sample set $\mathcal{E}$.

Algorithm 2: LCNFQ

input : the set of experience samples $\mathcal{E}$
output : approximated Q-function
initialize all neural nets B_{q_i} with (s_0, q_i, a) as the input and r_n as the output where $a \in \mathcal{A}$ is a random action
repeat
  for $q_i = |\mathcal{Q}|$ ***to*** 1 **do**
    $\mathcal{P}_{q_i} = \{(input_l, target_l),\ l = 1, ..., |\mathcal{E}_{q_i}|)\}$
      $input_l = ({s_l}^{\otimes}, a_l)$
      $target_l = R({s_l}^{\otimes}, a_l) + \gamma \max_{a'} Q({s_l}^{\otimes\prime}, a')$
      where $({s_l}^{\otimes}, a_l, {s_l}^{\otimes\prime}, R({s_l}^{\otimes}, a_l), q_i) \in \mathcal{E}_{q_i}$
    $B_{q_i} \leftarrow \text{Rprop}(\mathcal{P}_{q_i})$
  end
until *end of trial*

Once the exploration phase is completed and the sample set is created, learning is performed over the sample set. In the learning phase, we employ n separate multi-layer perceptrons, each with one hidden layer, where $n = |\mathcal{Q}|$ and $\mathcal{Q}$ is the finite cardinality of the automaton $\mathbf{N}$[5]. Each neural net is associated with a state in the LDBA and for each automaton state $q_i \in \mathcal{Q}$ the associated neural net is called $B_{q_i} : \mathcal{S}^{\otimes} \times \mathcal{A} \to \mathbb{R}$. Once the agent is at state $s^{\otimes} = (s, q_i)$ the neural net B_{q_i} is used for the local Q-function approximation. The set of neural nets acts as a global hybrid Q-function approximator $Q : \mathcal{S}^{\otimes} \times \mathcal{A} \to \mathbb{R}$. Note that the neural nets are not fully decoupled. For example, assume that by taking action a in state $s^{\otimes} = (s, q_i)$ the agent is moved to state $s^{\otimes\prime} = (s', q_j)$ where $q_i \neq q_j$. According to (4) the weights of B_{q_i} are updated such that $B_{q_i}(s^{\otimes}, a)$ has minimum possible error to $R(s^{\otimes}, a) + \gamma \max_{a'} B_{q_j}(s^{\otimes\prime}, a')$. Therefore, the value of $B_{q_j}(s^{\otimes\prime}, a')$ affects $B_{q_i}(s^{\otimes}, a)$.

Let $q_i \in \mathcal{Q}$ be a state in the LDBA. Then define $\mathcal{E}_{q_i} := \{(\cdot, \cdot, \cdot, \cdot, x) \in \mathcal{E} | x = q_i\}$ as the set of experiences within $\mathcal{E}$ that are associated to state q_i, i.e., $\mathcal{E}_{q_i}$ is the projection of $\mathcal{E}$ onto q_i. Once the experience set $\mathcal{E}$ is gathered, each neural net B_{q_i} is trained by its associated experience set $\mathcal{E}_{q_i}$. At each iteration a pattern set $\mathcal{P}_{q_i}$ is generated based on $\mathcal{E}_{q_i}$:

$$\mathcal{P}_{q_i} = \{(input_l, target_l), l = 1, ..., |\mathcal{E}_{q_i}|)\},$$

where $input_l = ({s_l}^{\otimes}, a_l)$ and $target_l = R({s_l}^{\otimes}, a_l) + \gamma \max_{a'} Q({s_l}^{\otimes\prime}, a')$ such that $({s_l}^{\otimes}, a_l, {s_l}^{\otimes\prime}, R({s_l}^{\otimes}, a_l), q_i) \in \mathcal{E}_{q_i}$. The pattern set is used to train the

[5] Different embeddings, such as one hot encoding [85] and integer encoding, have been applied in order to approximate the global Q-function with a single feedforward net. However, we have observed poor performance since these encodings allow the network to assume an ordinal relationship between automaton states. This means that by assigning integer numbers or one hot codes, automaton states are categorised in an ordered format, and can be ranked. Clearly, this disrupts Q-function generalisation by assuming that some states in product MDP are closer to each other. Consequently, we have turned to the use of n separate neural nets, which work together in a hybrid fashion, meaning that the agent can switch between these neural nets as it jumps from one automaton state to another.

neural net B_{q_i}. We use Rprop [86] to update the weights in each neural net, as it is known to be a fast and efficient method for batch learning [30]. In each cycle of LCNFQ (Algorithm 2), the training schedule starts from networks that are associated with accepting states of the automaton and goes backward until it reaches the networks that are associated to the initial states. In this way we allow the Q-value to back-propagate through the networks. In Algorithm 2, without loss of generality we assume that the automaton states are ordered and hence the back-propagation starts from $q_i = |\mathcal{Q}|$.

3.3 Alternatives to LCNFQ

In the following, we discuss the most popular alternative approaches to solving infinite-state MDPs, namely the Voronoi Quantiser (VQ) and Fitted Value Iteration (FVI). They will be benchmarked against LCNFQ in Section 4.

3.3.1 Voronoi Quantiser

VQ can be classified as a discretisation algorithm which abstracts the continuous-state MDP to a finite-state MDP, allowing classical RL to be run. However, most of discretisation techniques are usually done in an ad-hoc manner, disregarding one of the most appealing features of RL: autonomy. In other words, RL is able to produce the optimal policy with regards to the reward function, with minimum supervision. Therefore, the state-space discretisation should be performed as part of the learning task, instead of being fixed at the start of the learning process.

Inspired by [87], we propose a version of VQ that is able to discretise the state space of the product MDP $\mathcal{S}^{\otimes}$, while allowing RL to explore the MDP. VQ maps the state space onto a finite set of disjoint regions called Voronoi cells [88]. The set of centroids of these cells is denoted by $\mathcal{C} = \{c_i\}_{i=1}^{m},\ c_i \in \mathcal{S}^{\otimes}$, where m is the number of the cells. With $\mathcal{C}$, we are able to use QL and find an approximation of the optimal policy for a continuous-state space MDP.

In the beginning, $\mathcal{C}$ is initialised to consist of just one c_1, which corresponds to the initial state. This means that the agent views the entire state space as a homogeneous region when no a-priori knowledge is available. Subsequently, when the agent explores, the Euclidean distance between each newly visited state and its nearest neighbour is calculated. If this distance is greater than a threshold value Δ called "minimum resolution", or if the new state $s^{\otimes}$ comprises an automaton state that has never been visited, then the newly visited state is appended to $\mathcal{C}$. Therefore, as the agent continues to explore, the size of $\mathcal{C}$ would increase until the "relevant" parts of the state space are partitioned. In our algorithm, the set $\mathcal{C}$ has $|\mathcal{Q}|$ disjoint subsets where $\mathcal{Q}$ is the finite set of states of the automaton. Each subset $\mathcal{C}^{q_j},\ j = 1, ..., |\mathcal{Q}|$ contains the centroids of those Voronoi cells that have the form of $c_i^{q_j} = (\cdot, q_j)$, i.e. $\bigcup_i^m c_i^{q_j} = \mathcal{C}^{q_j}$ and $\mathcal{C} = \bigcup_{j=1}^{|\mathcal{Q}|} \mathcal{C}^{q_j}$. Therefore, a Voronoi cell

Algorithm 3: Episodic VQ

input : minimum resolution Δ
output : approximated Q-function Q

initialize $c_1 = c =$ initial state
initialize $Q(c_1, a) = 0,\ \forall a \in \mathcal{A}$
repeat
  set $\mathcal{C} = c_1$
  $\alpha = \arg\max_{a \in \mathcal{A}} Q(c, a)$
  repeat
    execute action α and observe the next state (s', q)
    if $\mathcal{C}^q$ *is empty* **then**
      append $c_{new} = (s', q)$ to $\mathcal{C}^q$
      initialize $Q(c_{new}, a) = 0,\ \forall a \in \mathcal{A}$
    else
      determine the nearest neighbour c_{new} within $\mathcal{C}^q$
      if $c_{new} = c$ **then**
        if $||c - (s', q)||_2 > \Delta$ **then**
          append $c_{new} = (s', q)$ to $\mathcal{C}^q$
          initialize $Q(c_{new}, a) = 0,\ \forall a \in \mathcal{A}$
        end
      else
        $Q(c, \alpha) = (1 - \mu)Q(c, \alpha) + \mu[R(c, \alpha) + \gamma \max_{a'}(Q(c_{new}, a'))]$
      end
    end
    $c = c_{new}$
  until *end of trial*
until *end of trial*

$$\{(s, q_j) \in \mathcal{S}^{\otimes}, ||(s, q_j) - c_i^{q_j}||_2 \leq ||(s, q_j) - c_{i'}^{q_j}||_2\},$$

is defined by the nearest neighbour rule for any $i' \neq i$. The proposed VQ algorithm is presented in Algorithm 3.

3.3.2 Fitted Value Iteration

FVI is an approximate DP algorithm for continuous-state MDPs, which employs function approximation techniques [89]. In standard DP the goal is to find a mapping, i.e. value function, from the state space to $\mathbb{R}$, which can generate the optimal policy. The value function in our setup is the expected reward in (1) when *Pol* is the optimal policy, i.e. U^{Pol^*}. Over continuous state spaces, analytical representations of the value function are in general not available. Approximations can be obtained numerically through approximate value iterations, which involve approximately iterating a Bellman operator on a an approximate value function [10].

We propose a modified version of FVI that can handle the product MDP. The global value function $v : \mathcal{S}^{\otimes} \to \mathbb{R}$, or more specifically $v : \mathcal{S} \times \mathcal{Q} \to \mathbb{R}$, consists of $|\mathcal{Q}|$ number of components. For each $q_j \in \mathcal{Q}$, the sub-value function

$v^{q_j} : \mathcal{S} \to \mathbb{R}$ returns the value the states of the form (s, q_j). Similar to the LCNFQ algorithm, the components are not decoupled.

Let $P^{\otimes}(dy|s^{\otimes}, a)$ be the distribution over $\mathcal{S}^{\otimes}$ for the successive state given that the current state is $s^{\otimes}$ and the selected action is a. For each state (s, q_j), the Bellman update over each component of value function v^{q_j} is defined as:

$$\tau v^{q_j}(s) = \sup_{a \in \mathcal{A}} \{ \int v(y) P^{\otimes}(dy|(s, q_j), a) \}, \tag{15}$$

where τ is the Bellman operator [90]. The update in (15) is different than the standard Bellman update in DP, as it does not comprise a running reward, and as the (terminal) reward is replaced by the following function initialization:

$$v(s^{\otimes}) = \begin{cases} r_p & \text{if } s^{\otimes} \in \mathbb{A}, \\ r_n & \text{otherwise.} \end{cases} \tag{16}$$

Here r_p and r_n are defined in (8) with $y = 0$. The main hurdle in executing the Bellman operator in continuous state MDPs, as in (15), is that no analytical representations of the value function v and of its components v^{q_j}, $q_j \in \mathcal{Q}$ are in general available. Therefore, we employ an approximation method, by introducing a new operator L.

The operator L provides an approximation of the value function, denoted by Lv, and of its components v^{q_j}, which we denote by Lv^{q_j}. For each $q_j \in \mathcal{Q}$ the approximation is based on a set of points $\{(s_i, q_j)\}_{i=1}^{k} \subset \mathcal{S}^{\otimes}$ which are called centres. For each q_j, the centres $i = 1, ..., k$ are distributed uniformly over $\mathcal{S}$.

We employ a kernel-based approximator for our FVI algorithm. Kernel-based approximators have attracted a lot of attention mostly as they perform very well in high-dimensional state spaces [10]. One of these methods is the kernel averager, which can be represented by the following expression for each state (s, q_j):

$$Lv(s, q_j) = Lv^{q_j}(s) = \frac{\sum_{i=1}^{k} K(s_i - s) v^{q_j}(s_i)}{\sum_{i=1}^{k} K(s_i - s)}, \tag{17}$$

where the kernel $K : \mathcal{S} \to \mathbb{R}$ is a radial basis function, such as $e^{-|s-s_i|/h'}$, and h' is smoothing parameter. Each kernel is characterised by the point s_i, and its value decays to zero as s diverges from s_i. This means that for each $q_j \in \mathcal{Q}$ the approximation operator L in (17) is a convex combination of the values of the centres $\{s_i\}_{i=1}^{k}$ with larger weight given to those values $v^{q_j}(s_i)$ for which s_i is close to s. Note that the smoothing parameter h' controls the weight assigned to more distant values.

In order to approximate the integral in the Bellman update (15) we use a Monte Carlo sampling technique [91]. For each centre (s_i, q_j) and for each action a, we sample the next state $y_a^z(s_i, q_j)$ for $z = 1, ..., Z$ times and append

Algorithm 4: FVI

input : MDP $\mathbf{M}$, a set of samples $\{s_i^{\otimes}\}_{i=1}^{k} = \{(s_i, q_j)\}_{i=1}^{k}$ for each $q_j \in \mathcal{Q}$, Monte Carlo sampling number Z, smoothing parameter h'
output : approximated value function Lv

initialize Lv
sample $\mathcal{Y}_a^Z(s_i, q_j),\ \forall q_j \in \mathcal{Q},\ \forall i = 1, ..., k\ ,\ \forall a \in \mathcal{A}$
repeat
  for $j = |\mathcal{Q}|$ ***to*** 1 **do**
    $\forall q_j \in \mathcal{Q},\ \forall i = 1, ..., k\ ,\ \forall a \in \mathcal{A}$ calculate $I_a((s_i, q_j)) = 1/Z \sum_{y \in \mathcal{Y}_a^Z(s_i, q_j)} Lv(y)$ using (17)
    for each state (s_i, q_j), update $v^{q_j}(s_i) = \sup_{a \in \mathcal{A}} \{I_a((s_i, q_j))\}$ in (17)
  end
until *end of trial*

these samples to the set of Z subsequent states $\mathcal{Y}_a^Z(s_i, q_j)$. We then replace the integral with

$$I_a(s_i, q_j) = \frac{1}{Z} \sum_{z=1}^{Z} Lv(y_a^z(s_i, q_j)). \tag{18}$$

The approximate value function Lv is initialised according to (16). In each cycle of FVI, the approximate Bellman update is first performed over the sub-value functions that are associated with accepting states of the automaton, i.e. those that have initial value of r_p, and then goes backward until it reaches the sub-value functions that are associated to the initial states. In this manner, we allow the state values to back-propagate through the transitions that connects the sub-value function via (18). Without loss of generality we assume that the automaton states are ordered and hence the back-propagation starts from $q_i = |\mathcal{Q}|$. Once we have the approximated value function, we can generate the optimal policy by following the maximum value (Algorithm 4).

We conclude this section by emphasising that Algorithms 3 and 4 are proposed to be benchmarked against LCNFQ, later in Section 4. Further, MDP abstraction techniques such as [7] failed to scale and to find an optimal policy. In the following we describe another contribution of this work in dealing with time-varying MDPs that show periodic behaviours. Such behaviours can be seen in a number of applications such as physical systems and video games.

3.4 Transfer Learning for Time-Varying MDPs

The classical RL settings of static MDPs are not particularly realistic abstractions of the real world where the environment is not completely static. To this end, we consider MDPs that exhibit time-dependent behaviour. The idea to adopt this was inspired by the initial chamber of Atari 2600 Montezuma's Revenge, where a skull rolls periodically on a platform. Such time-varying obstacle can easily render QL useless as the method is memory-less and only takes into consideration the current state when choosing the best action.

Periodic behaviour can be encompassed in the MDP dynamics by extending the state space with a time variable. However, this approach is in general not computationally viable due to state-space explosion, which is caused by adding the ever-extending extra dimension of time. More specifically, for each time-step in the period of the MDP dynamics, the agent finds itself in a completely new environment and has to learn everything from scratch.

To overcome this limitation we model the periodically moving obstacle as a Kripke structure. The structure over $\mathcal{AP}^{\otimes}$, which comprises the possible positions of the obstacle, when unfolded to account for directionality (as shown in Fig. 1), represents the obstacle positions. It can be defined as $\mathbf{D}(\mathcal{K}, k_0, \Delta', L')$ where $\mathcal{K}$ is the Kripke state space with transition relation $\Delta' \subseteq \mathcal{K} \otimes \mathcal{K}$ and labelling function $L' : \mathcal{K} \to 2^{\mathbb{N}}$. In this paper, for the sake of simplicity, we assume that the labelling function assigns a natural number to each state.

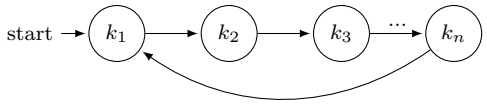


Fig. 1: Kripke structure capturing the time-varying behaviour of the MDP.

We propose to first take the cross product with the generated LDBA $\mathbf{N}$ (Definition 11) and Kripke structure $\mathbf{D}$. The resulting structure is a time-varying automaton with which we can synchronise the original MDP $\mathbf{M}$ on-the-fly (see Remark 3). Thanks to this new automaton and the fact that no explicit product MDP is constructed, the proposed approach can handle time-dependent behaviour of the MDP much more efficiently as opposed to explicit encoding of time as a lifting to the MDP.

The final product captures the time-varying parts of the original MDP, and maps it to a static MDP including $\mathbf{D}$. The following definition formalises this intuition:

Definition 18 (Periodic Product MDP) Given an MDP $\mathbf{M}(\mathcal{S}, \mathcal{A}, s_0, P, \mathcal{AP}, L)$, an LDBA $\mathbf{N}(\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ with $\Sigma = 2^{\mathcal{AP}}$, and a Kripke structure $\mathbf{D}(\mathcal{K}, k_0, \Delta', L')$, the product MDP is defined as $\mathbf{M_{ND}}(\mathcal{S}^{\otimes}, \mathcal{A}, s_0^{\otimes}, P^{\otimes}, \mathcal{AP}^{\otimes}, L^{\otimes}, \mathcal{F}^{\otimes})$, where $\mathcal{S}^{\otimes} = \mathcal{S} \times (\mathcal{Q} \times \mathcal{K})$, $s_0^{\otimes} = (s_0, q_0, k_0)$, $\mathcal{AP}^{\otimes} = \mathcal{Q}$, $L^{\otimes} : \mathcal{S}^{\otimes} \to 2^{\mathcal{Q}}$ such that $L^{\otimes}(s, q, k) = q$ and $\mathcal{F}^{\otimes} \subseteq \mathcal{S}^{\otimes}$ is the set of accepting states $\mathcal{F}^{\otimes} = \{F_1^{\otimes}, ..., F_f^{\otimes}\}$ where $F_j^{\otimes} = \mathcal{S} \times F_j \times \mathcal{K}$. The intuition behind the transition kernel $P^{\otimes}$ is that given the current state (s_i, q_i, k_i) and action a, the new state is (s_j, q_j, k_j) where $s_j \sim P(\cdot|s_i, a)$, $q_j \in \Delta(q_i, L(s_j))$, and $k_j \in \Delta'(k_i)$. When the MDP $\mathbf{M}$ is finite-state then $P^{\otimes} : \mathcal{S}^{\otimes} \times \mathcal{A} \times \mathcal{S}^{\otimes} \to [0, 1]$ is the transition probability function such that $(s_i \xrightarrow{a} s_j) \wedge (q_i \xrightarrow{L(s_j)} q_j) \wedge (k_i \to k_j) \Rightarrow P^{\otimes}((s_i, q_i, k_i), a, (s_j, q_j, k_j)) = P(s_i, a, s_j)$.

The curse of dimensionality related to the alternative explicit encoding of the periodicity within the MDP is now turned into slowness of the learning

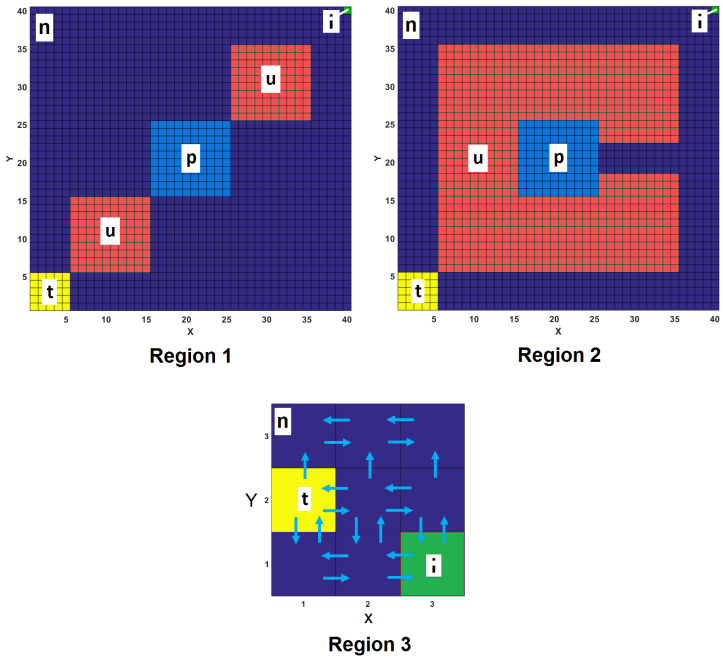


Fig. 2: MDP labelling - green: initial state (i); dark blue: neutral (n); red: unsafe (u); light blue: pre-target (p); yellow: target (t).

process. To speed up the process we can use the observation that the agent, and inherently the Q-values, are only affected by the time-varying parts of the original MDP when they are in close proximity. In other words, the optimal strategy can be learnt more quickly by sharing the state-action values across the dimension defined by $\mathcal{K}$.

Recall the classical update rule in QL, when the agent executes action a at state $s^{\boxtimes} = (s, q, k)$ is as follows:

$$Q(s^{\boxtimes}, a) \leftarrow Q(s^{\boxtimes}, a) + \mu[R(s^{\boxtimes}, a) + \gamma \max_{a' \in \mathcal{A}}(Q(s^{\boxtimes\prime}, a')) - Q(s^{\boxtimes}, a)],$$

Once a positive behaviour is learned in dealing with the time-varying part, it is propagated along the $\mathcal{K}$ dimension to allow the agent to do the same behaviour in states that are in close proximity of the time-varying part. Technically speaking, when the agent executes action a at state $s^{\boxtimes} = (s, q, k)$ and updates the Q-value for $s^{\boxtimes}, a$, then

$$\forall k' \in \mathcal{K} \setminus \{k\},\ \ Q(s, q, k', a) \leftarrow \max\{Q(s, q, k, a), Q(s, q, k', a)\}.$$

This update rule, once combined with QL classic update rule, allows the positive behaviours to be echoed in $\mathcal{S}^{\boxtimes}$ and significantly reduces the learning time.

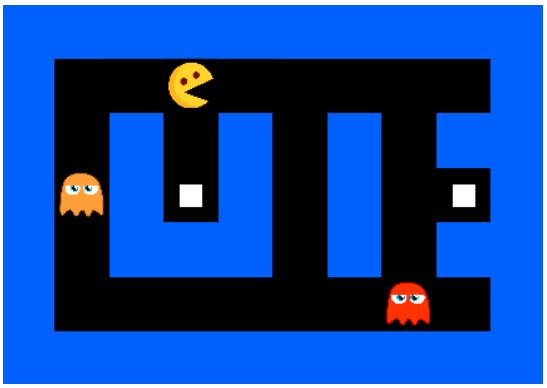


Fig. 3: Pacman environment (initial condition) - the square on the left is labelled as food 1 (f_1) and the one on the right as food 2 (f_2), the state of being caught by a ghost is labelled as (g) and the rest of the state space is neutral (n).

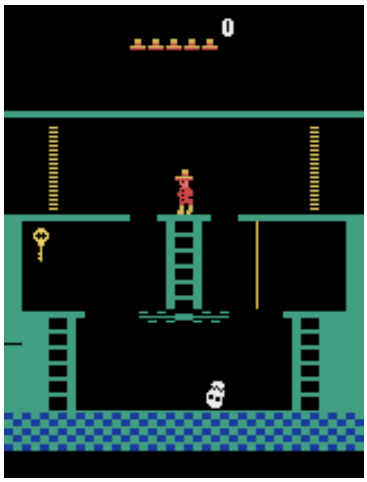


Fig. 4: Initial condition in the first level of Montezuma's Revenge.

4 Experimental Results

We discuss a number of planning experiments dealing with policy synthesis problems around temporal specifications that are extended with safety requirements, both when the state-space is finite and continuous.

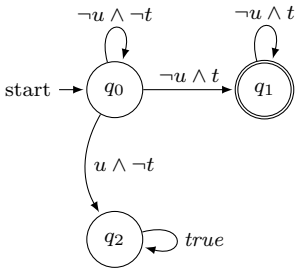


Fig. 5: LDBA for the specification in (19) with removed transitions labelled $t \wedge u$ (since it is impossible to be at target and unsafe at the same time).

4.1 Finite-state MDPs

4.1.1 Models

The first experiment is an LTL-constrained control synthesis problem for a robot in a slippery grid-world. Let the grid be an $L \times L$ square over which the robot moves. In this setup, the robot location is the MDP state $s \in \mathcal{S}$. At each state $s \in \mathcal{S}$ the robot has a set of actions $\mathcal{A} = \{\mathit{left}, \mathit{right}, \mathit{up}, \mathit{down}, \mathit{stay}\}$ by which the robot is able to move to other states (e.g. s') with the probability of $P(s, a, s'), a \in \mathcal{A}$. At each state $s \in \mathcal{S}$, the actions available to the robot are either to move to a neighbour state $s' \in \mathcal{S}$ or to stay at the state s. In this example, if not otherwise specified, we assume for each action the robot chooses, there is a probability of 85% that the action takes the robot to the correct state and 15% that the action takes the robot to a random state in its neighbourhood (including its current state). This example is a well-known benchmark and is often referred to as "slippery grid-world".

A labelling function $L : \mathcal{S} \to 2^{\mathcal{AP}}$ assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \subseteq \mathcal{AP}$. We assume that in each state s the robot is aware of the labels of the neighbouring states. We consider two 40×40 regions and one 3×3 region with different labels as in Fig. 2. In Region 3 and in the state target, the subsequent state after performing action *stay* is always the state target itself. Note that all the actions are not active in Region 3 and the agent has to avoid the top row otherwise it gets trapped.

The second experiment is a version of the well-known Atari 2600 game Pacman, which is initialised in a tricky configuration (Fig. 3). In order to win the game the agent has to collect all available tokens without being caught by moving ghosts. The ghost dynamics is stochastic: a probability p_g for each ghost determines if the ghost is chasing Pacman (often referred to as "chase mode") or if it is executing a random action ("scatter mode"). Notice that, unlike the first experiment, in this setup the actions of the ghosts and of the agent result in a deterministic transition, i.e. the world is not "slippery".

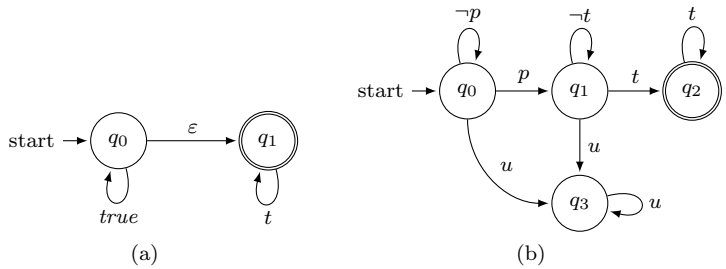


Fig. 6: (a) LDBA for the specification in (20) - (b) LDBA for property in (21).

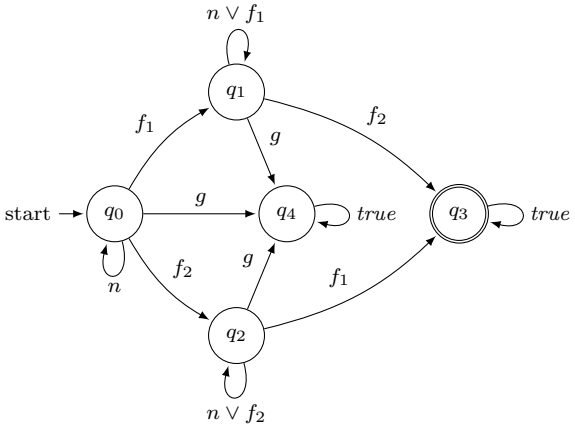


Fig. 7: LDBA for the specification in (22).

The third experiment deals with the complex environment of Atari 2600 Montezuma's Revenge. To win the game the agent needs to descend down the ladders, to jump over the skull, to fetch the key and to return to the top and to open one of the doors. In this experiment, we assume that for each action that the agent selects, there is a 90% that the chosen action is executed and a 10% that a random action is instead performed.

Using the OpenAI gym environment [92] we set as our test-bed the first chamber of Montezuma's Revenge (Fig. 4). In order to enable a tabular approach, we reduce the size of the state-action space. This simplifications is only a means of expediting the training process, but no generality is lost since the algorithm could work just as well with the full set of actions, except it would require additional time and resources. The goal is to achieve a better score overall than some modern algorithms, such as the DQN approach, which on average has achieved a score of zero [32].

In the following, we describe the methodology of simplifying the environment to make the testing feasible. The game simulator is very general and therefore comes with 18 possible actions, many of which do not apply in our environment,

e.g. "FIRE". We conclude that using 6 actions for movement out of the 18 are sufficient, only excluding the actions relating to firing and that of doing nothing. We extract and work with pixel matrix to identify and locate different elements of the state space. Hence, we treat the state space as being discrete.

4.1.2 Specifications

In the first experiment (slippery grid-world), we consider the following LTL properties. The two first properties (19) and (20) focus on safety and reachability while the third property (21) requires a sequential visit to states with label p and then target t:

$$\Diamond t \wedge \Box(t \rightarrow \Box t) \wedge \Box(u \rightarrow \Box u), \tag{19}$$

$$\Diamond \Box t, \tag{20}$$

and

$$\Diamond(p \wedge \Diamond t) \wedge \Box(t \rightarrow \Box t) \wedge \Box(u \rightarrow \Box u), \tag{21}$$

where t stands for "target", u stands for "unsafe", and p refers to the area that has to be visited before visiting the area with label t. Property (19) asks the agent to eventually find the target $\Diamond t$ and to stay there $\Box(t \rightarrow \Box t)$, while avoiding the unsafe - otherwise it is going to be trapped there $\Box(u \rightarrow \Box u)$. Specification (20) requires the agent to eventually find the target and to stay there. The intuition behind (21) is that the agent has to eventually first visit p and then visit t at some point in the future $\Diamond(p \wedge \Diamond t)$ and stay there $\Box(t \rightarrow \Box t)$ while avoiding unsafe areas $\Box(u \rightarrow \Box u)$.

The LDBAs associated with (19), (20) and (21) are in Fig. 5, and Fig. 6 respectively. Note that the LDBA expressing (19) in Fig. 5 is deterministic and only needs the state set $\mathcal{Q}_D$ as in Definition 10 while the LDBA in Fig. 6. a needs both $\mathcal{Q}_D$ and $\mathcal{Q}_N$ to express (20).

In the second experiment (Pacman), in order to win the game, Pacman is required to choose between one of the two available foods and then find the other one ($\Diamond[(f_1 \wedge \Diamond f_2) \vee (f_2 \wedge \Diamond f_1)]$) while avoiding unwelcome shapes on the display, i.e. ghosts ($\Box(g \rightarrow \Box g)$). These clauses are what a human can perceive just by looking at the game screen and we feed a conjunction of these associations to the agent by using the following LTL formula:

$$\Diamond[(f_1 \wedge \Diamond f_2) \vee (f_2 \wedge \Diamond f_1)] \wedge \Box(g \rightarrow \Box g), \tag{22}$$

The constructed LDBA is shown in Fig. 7.

The third experiment (Montezuma's Revenge) is a rather more complicated environment, where the probability of winning the game by randomly exploring the state space is close to zero. However, a human player can derive the logical property behind the game by simply looking at the map and associating the acquiring of the key to the ability of opening the door $\Diamond(k \wedge \Diamond d)$ and staying at the door to win the first chamber $\Box(d \rightarrow \Box d)$, while avoiding

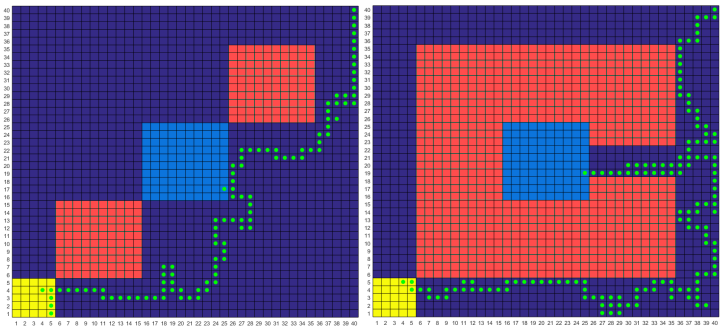


Fig. 8: Simulation results for specification (21) in (a) Region 1 and (b) Region 2.

the moving skull $\Box(s \rightarrow \Box s)$. Here k represent the key, d is the door, and s is the skull. We can then combine these constraints and express them as an LTL formula, such as:

$$\Diamond(k \wedge \Diamond d) \wedge \Box(d \rightarrow \Box d) \wedge \Box(s \rightarrow \Box s), \tag{23}$$

Note that this LTL formula is equivalent to (21) and therefore we employ the LDBA in Fig. 6, however with different labels. The LDBA built from the formula encompasses the safety and the goal of the agent, however to deal with the moving skull we represent the location and direction of the skull as a Kripke structure, which allows the agent to learn Q-values that generate policies avoiding the danger.

As mentioned earlier, the probability of randomly reaching the key and moving back to the door is very low, and even advanced algorithms, such as DQN [32], fail to achieve the overall goal. Of course, human intuition can solve this game and synthesise a successful policy – the advantage of our automated synthesis technique is that it does not require complete end-to-end examples, and thus it may avoid local optima that the human may believe to be global.

Note that the agent also loses a life when it falls from a high enough altitude, but this is not something that we can concretely assume and as such, it will not be penalised. Since we do not control the game engine, the agent will continue to lose its life upon falling, but rather than actively avoiding these moves, the agent will simply learn that the Q-value is null and there are better actions to be chosen.

4.1.3 Simulation Results

In the first experiment (slippery grid-world), the simulation parameters are set as $\mu = 0.9$ and $\gamma = 0.9$. Fig. 8 gives the results of the learning for the expression (21) in Region 1 and Region 2 after $400,000$ iterations and 200 learning episodes. Again, according to (21) the robot has to avoid the red

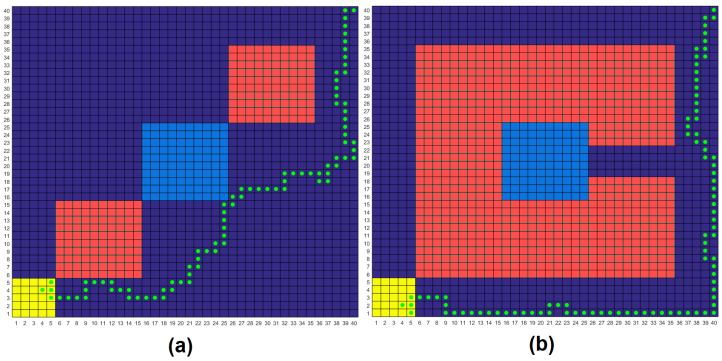


Fig. 9: Simulation results for specification (19) in (a) Region 1 and in (b) Region 2.

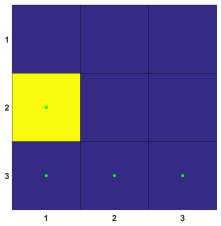


Fig. 10: Simulation results for (20) in Region 3.

(unsafe) regions, visit the light-blue (pre-target) area at least once and then go to the yellow (target) region. Recall that selected action is executed with a probability of 85%. Thus, there might be some undesired deviations in the robot path.

Fig. 9 gives the results of the learning for the LTL formula (19) in Region 1 and Region 2 after $400,000$ iterations and 200 learning episodes. The intuition behind the LTL formula in (19) is that the robot has to avoid red (unsafe) areas until it reaches the yellow (target) region, otherwise the robot is going to be stuck in the red (unsafe) area.

Finally, in Fig. 10 the learner tries to satisfy the LTL formula $\Diamond\Box t$ in (20). The learning takes 1000 iterations and 20 learning episodes.

Fig. 11 gives the result of our proposed value iteration method for calculating the maximum PSP in Region 2 with (21) and Region 3 with (20). In both cases our method was able to accurately calculate the maximum probability of satisfying the LTL property. We observed a monotonic decrease in the maximum error between the correct PSP calculated by PRISM and the probability calculation by LCQL (Fig. 12.a). Fig. 12.b shows the distance that agent traverses from initial state to final state at each learning episode in Region 1 under (21). After almost 400 episodes of learning the agent converges to the final optimal policy and the travelled distance stabilizes.

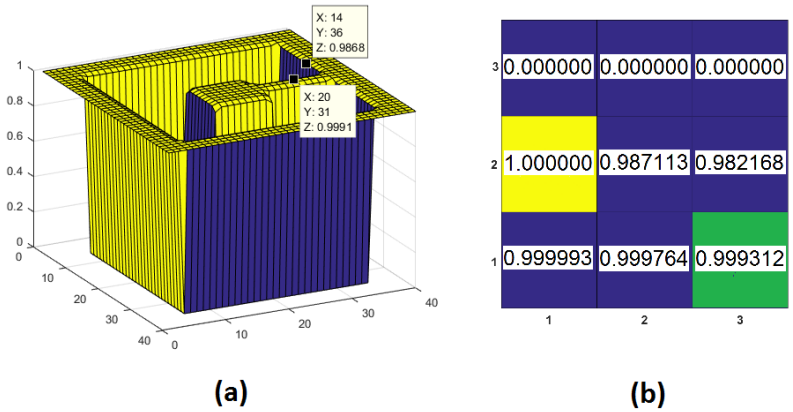


Fig. 11: PSP in (a) Region 2 with property (21) and in (b) Region 3 with specification (20). The results generated by LCQL are identical to the outcomes from PRISM.

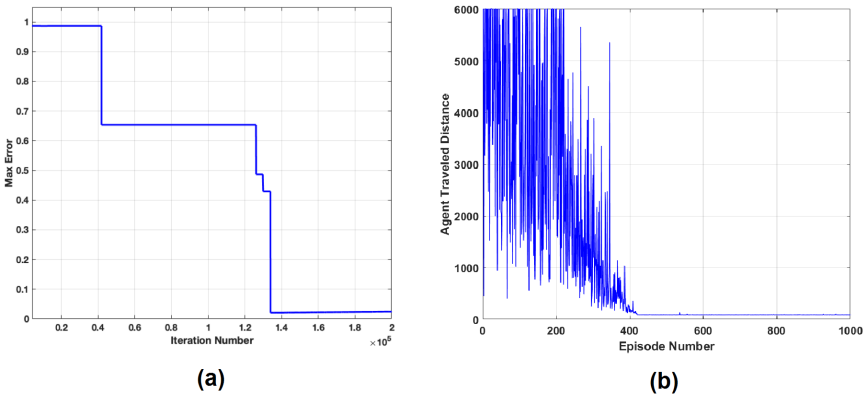


Fig. 12: In Region 2 under specification (21): (a) Maximum error between PSP computed with LCQL and with PRISM (b) The distance that agent traverses from initial state to final state with LCQL.

In the second experiment (Pacman), the simulation parameters are set as $\mu = 0.9$ and $\gamma = 0.9$. The stochastic behaviour of the ghosts is also captured by $p_g = 0.9$. Fig. 13 gives the results of learning with LCQL[6] and classical RL for (7). After almost 20,000 episodes, LCQL finds a stable policy to win the game even with ghosts playing probabilistically. The average steps for the agent to win the game (y axis) in LCQL is around 25 steps, which is very close to the human-level performance of 23 steps if the ghosts act

[6] Please visit https://www.cs.ox.ac.uk/conferences/lcrl to watch the videos of the agent playing Pacman. The code is adaptive and can be run under new configurations.

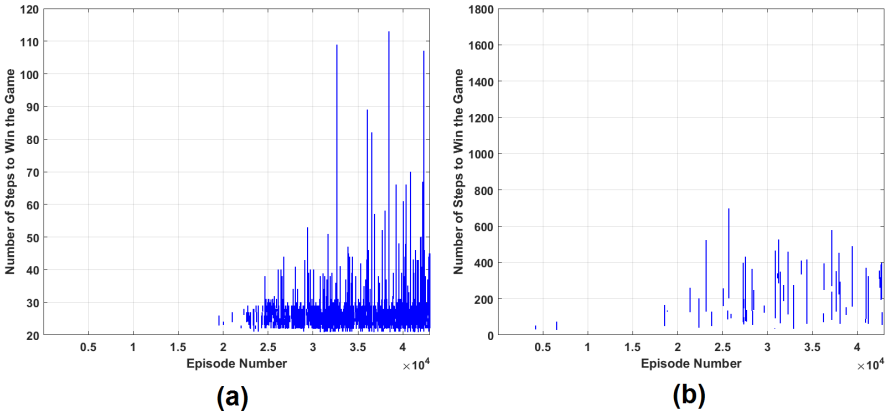


Fig. 13: Results of learning in Pacman with (a) LCQL, and (b) classical QL with positive reward for winning the game.

Fig. 14: Montezuma's Revenge - The agent successfully unlocks the door (notice reward).

deterministically. On the other hand, standard RL (in this case, classical QL with positive reward for winning the game) fails to find a stable policy and only achieves a number of random winnings with associated large numbers of steps.

In the third experiment (Montezuma's Revenge) the agent succeeds in reaching the set target after relatively short training (10000 episodes), which on a machine with a 3.2GHz Core i5 processor and 8GB of RAM, running Windows 7 took over a day, producing the results shown on Figure 14. The simulation parameters are set as $\mu = 0.75$, $\gamma = 0.9$.

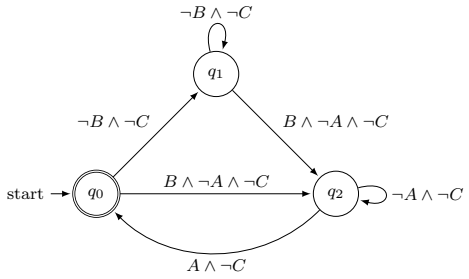


Fig. 15: LDBA expressing the LTL formula in (24) with removed transitions labelled $A \wedge B$ (since it is impossible to be at A and B at the same time).

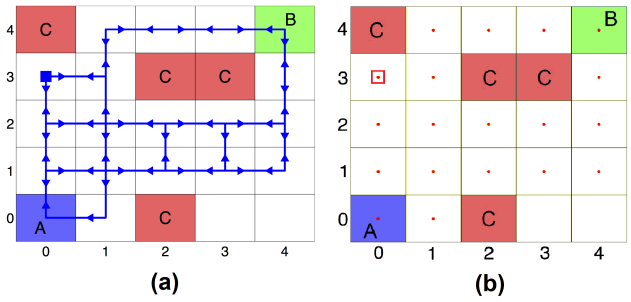


Fig. 16: (a) Example considered in [17]. (b) Trajectories under the policy generated by LCQL in [17].

4.1.4 Comparison with a DRA-based Learning Algorithm

The problem of LTL-constrained learning is also investigated in [17], where the authors propose to translate the LTL property into a DRA and then to construct a product MDP. A 5×5 grid world is considered and starting from state $(0, 3)$ the agent has to visit two regions infinitely often (areas A and B in Fig. 16). The agent has to also avoid the area C. This property can be encoded as the following LTL formula:

$$\Box\Diamond A \wedge \Box\Diamond B \wedge \Box\neg C. \tag{24}$$

The product MDP in [17] contains 150 states, which means that the Rabin automaton has 6 states. Fig. 16.a shows the trajectories under the optimal policy generated by [17] algorithm after 600 iterations. However, by employing LCQL we are able to generate the same trajectories with only 50 iterations (Fig. 16.b). The automaton that we consider is an LDBA with only 3 states as in Fig. 15. This result in a smaller product MDP and a much more succinct state space (only 75 states) for the algorithm to learn, which consequently leads to a faster convergence.

In addition, the reward shaping in LCQL is significantly simpler thanks to the Büchi acceptance condition. In a DRA $\mathbf{R}(\mathcal{Q}, \mathcal{Q}_0, \Sigma, \mathcal{F}, \Delta)$, the set $\mathcal{F} = \{(G_1, B_1), \ldots, (G_{n_F}, B_{n_F})\}$ represents the acceptance condition in which G_i, $B_i \in \mathcal{Q}$ for $i = 1, \ldots, n_F$. An infinite run $\theta \in \mathcal{Q}^\omega$ starting from $\mathcal{Q}_0$ is accepting if there exists $i \in \{1, \ldots, n_F\}$ such that

$$inf(\theta) \cap G_i \neq \emptyset \quad \text{and} \quad inf(\theta) \cap B_i = \emptyset.$$

Therefore for each $i \in \{1, \ldots, n_F\}$ a separate reward assignment is needed in [17] which complicates the implementation and increases the required calculation costs. This complicated reward assignment is not needed by employing the accepting frontier function in our framework.

More importantly, LCQL is a model-free learning algorithm that does not require an approximation of the transition probabilities of the underlying MDP. This even makes LCQL more easier to employ. We would like to emphasize that LCQL convergence proof solely depends on the structure of the MDP and this allows LCQL to find satisfying policies even if they have probability of less than one.

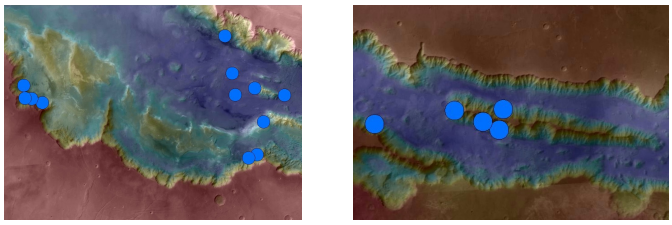
4.2 Continuous-state MDPs

In this section, we describe a mission planning architecture for an autonomous Mars-rover that uses LCNFQ to follow a mission on Mars. The scenario of interest is that we start with an image from the surface of Mars and then we add the desired labels from $2^{\mathcal{AP}}$, e.g. safe or unsafe, to the image. We assume that we know the highest possible disturbance caused by different factors (such as sand storms) on the rover motion. This assumption can be set to be very conservative given the fact that there might be some unforeseen factors that we did not take into account.

The next step is to express the desired mission in LTL format and run LCNFQ on the labelled image before sending the rover to Mars. We would like the rover to satisfy the given LTL property with the highest probability possible starting from any random initial state (as we can not predict the landing location exactly). Once LCNFQ is trained we use the network to guide the rover on the Mars surface. We compare LCNFQ with Voronoi quantizer and FVI and we show that LCNFQ outperforms these methods.

4.2.1 MDP Structure

In this numerical experiment the area of interest on Mars is Coprates quadrangle, which is named after the Coprates River in ancient Persia. There exist a significant number of signs of water, with ancient river valleys and networks of stream channels showing up as sinuous and meandering ridges and lakes. We consider two parts of Valles Marineris, a canyon system in Coprates quadrangle (Fig. 17). The blue dots, provided by NASA, indicate locations of recurring slope lineae (RSL) in the canyon network. RSL are seasonal dark streaks regarded



(a) Melas Chasma (b) Coprates Chasma

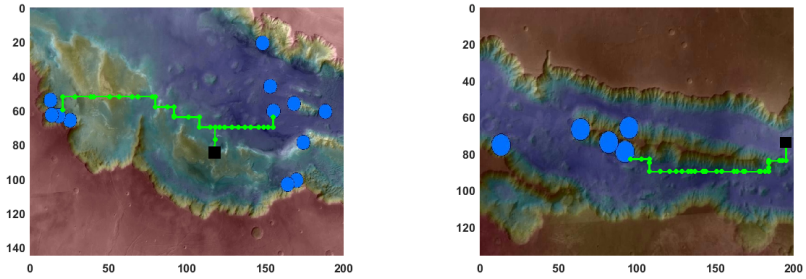
Fig. 17: Melas Chasma and Coprates Chasma, in the central and eastern portions of Valles Marineris. Map colour spectrum represents elevation, where red is high and blue is low. (Image courtesy of NASA, JPL, Caltech and University of Arizona.)

as the strongest evidence for the possibility of liquid water on the surface of Mars. RSL extend down-slope during a warm season and then disappear in the colder part of the Martian year [93]. The two areas mapped in Fig. 17, Melas Chasma and Coprates Chasma, have the highest density of known RSL.

For each case, let the entire area be our MDP state space $\mathcal{S}$, where the rover location is a single state $s \in \mathcal{S}$. At each state $s \in \mathcal{S}$, the rover has a set of actions $\mathcal{A} = \{\mathit{left}, \mathit{right}, \mathit{up}, \mathit{down}, \mathit{stay}\}$ by which it is able to move to other states: at each state $s \in \mathcal{S}$, when the rover takes an action $a \in \{\mathit{left}, \mathit{right}, \mathit{up}, \mathit{down}\}$ it is moved to another state (e.g., s') towards the direction of the action with a range of movement that is randomly drawn from $(0, D]$ unless the rover hits the boundary of the area which forces the rover to remain on the boundary. In the case when the rover chooses action $a = \mathit{stay}$ it is again moved to a random place within a circle centred at its current state and with radius $d \ll D$. Again, d captures disturbances on the surface of Mars and can be tuned accordingly.

With $\mathcal{S}$ and $\mathcal{A}$ defined we are only left with the labelling function $L : \mathcal{S} \to 2^{\mathcal{AP}}$ which assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \subseteq 2^{\mathcal{AP}}$. With the labelling function, we are able to divide the area into different regions and define a logical property over the traces that the agent generates. In this particular experiment, we divide areas into three main regions: neutral, unsafe and target. The target label goes on RSL (blue dots), the unsafe label lays on the parts with very high elevation (red coloured) and the rest is neutral. In this example we assume that the labels do not overlap each other.

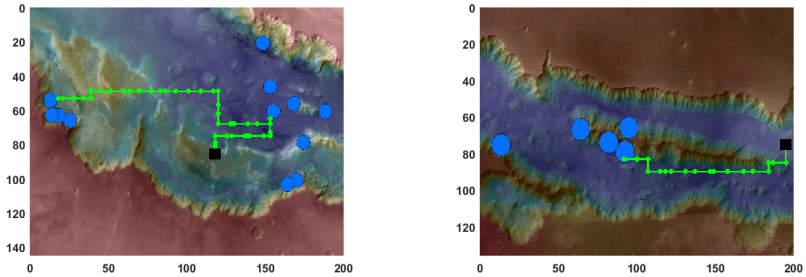
Note that when the rover is deployed to its real mission, the precise landing location is not known. Therefore, we should take into account the randomness of the initial state s_0. The dimensions of the area of interest in Fig. 17.a are 456.98×322.58 km and in Fig. 17.b are 323.47×215.05 km. The diameter of each RSL is 19.12 km. Other parameters in this numerical example have been set as $D = 2$ km, $d = 0.02$ km, the reward function parameter $y = 1$ for LCNFQ and $y = 0$ for VQ and FVI, $M = 1$, $m = 0.05$ and $\mathcal{AP} = \{$neutral, unsafe, target_1, target_2$\}$.



(a) Melas Chasma and landing location (black rectangle) (118, 85)

(b) Coprates Chasma and landing location (black rectangle) (194, 74)

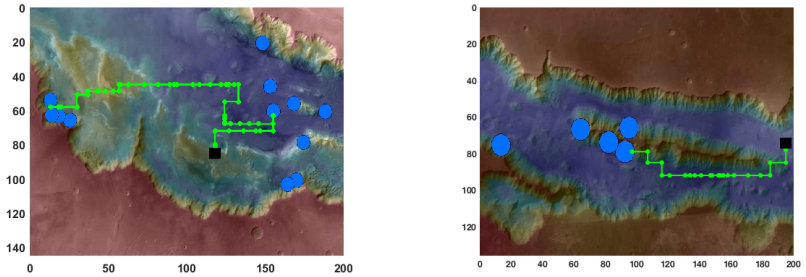
Fig. 18: Generated paths by LCNFQ.



(a) Melas Chasma and landing location (black rectangle) (118, 85)

(b) Coprates Chasma and landing location (black rectangle) (194, 74)

Fig. 19: Generated paths by episodic VQ.



(a) Melas Chasma and landing location (black rectangle) (118, 85)

(b) Coprates Chasma and landing location (black rectangle) (194, 74)

Fig. 20: Generated paths by FVI.

Table 1: Simulation results for Continuous-State MDPs

Melas Chasma					
Algorithm	**Sample Complexity**	$U^{Pol^*}(s_0)$	**Success Rate**†	**Training Time*****(s)**	**Iteration Num.**
LCNFQ	**7168** samples	**0.0203**	**99%**	95.64	**40**
VQ ($\Delta = 0.4$)	27886 samples	0.0015	**99%**	1732.35	2195
VQ ($\Delta = 1.2$)	7996 samples	0.0104	97%	273.049	913
VQ ($\Delta = 2$)	-	0	0%	-	-
FVI	40000 samples	0.0133	98%	**4.12**	80
Coprates Chasma					
Algorithm	**Sample Complexity**	$U^{Pol^*}(s_0)$	**Success Rate**†	**Training Time*****(s)**	**Iteration Num.**
LCNFQ	**2680 samples**	**0.1094**	**98%**	166.13	**40**
VQ ($\Delta = 0.4$)	8040 samples	0.0082	**98%**	3666.18	3870
VQ ($\Delta = 1.2$)	3140 samples	0.0562	96%	931.33	2778
VQ ($\Delta = 2$)	-	0	0%	-	-
FVI	25000 samples	0.0717	97%	**2.16**	80

† Testing the trained agent (for 100 trials) * Average for 10 trainings

4.2.2 Specifications

The first control objective in this numerical example is expressed by the following LTL formula over Melas Chasma (Fig. 17.a):

$$\Diamond(p \wedge \Diamond t) \wedge \Box(t \rightarrow \Box t) \wedge \Box(u \rightarrow \Box u), \tag{25}$$

where n stands for "neutral", p stands for "target 1", t stands for "target 2" and u stands for "unsafe". Target 1 are the RSL (blue bots) on the right with a lower risk of the rover going to unsafe region and the target 2 label goes on the left RSL that are a bit riskier to explore. Conforming to (25) the rover has to visit the target 1 (any of the right dots) at least once and then proceed to the target 2 (left dots) while avoiding unsafe areas. Note that according to $\Box(u \rightarrow \Box u)$ in (25) the agent is able to go to unsafe area u (by climbing up the slope) but it is not able to come back due to the risk of falling. Note that the LDBA expressing (25) is as in Fig. 6.a.

The second formula focuses more on safety and we are going to employ it in exploring Coprates Chasma (Fig. 17.b), where a critical unsafe slope exists in the middle of this region:

$$\Diamond t \wedge \Box(t \rightarrow \Box t) \wedge \Box(u \rightarrow \Box u) \tag{26}$$

Here, t refers to the "target", i.e. RSL in the map, and u stands for "unsafe". According to this LTL formula, the agent has to eventually reach the target ($\Diamond t$) and stays there ($\Box(t \rightarrow \Box t)$). However, if the agent hits the unsafe area it can never comes back and remains there forever ($\Box(u \rightarrow \Box u)$). With (26) we can again build the associated Büchi automaton as in Fig. 6.b. Having the Büchi automaton for each formula, we are able to use Definition 11 to build product MDPs and run LCNFQ on both.

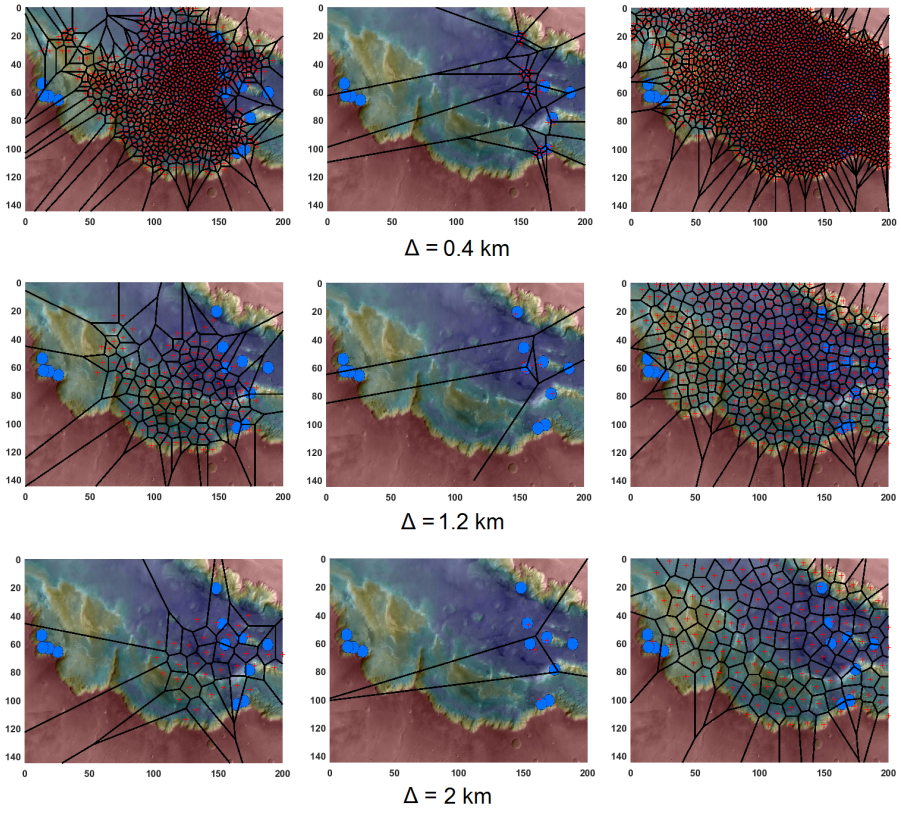


Fig. 21: VQ – generated cells in Melas Chasma for different resolutions.

4.2.3 Simulation Results

This section presents the simulation results. All simulations are carried on a machine with a 3.2GHz Core i5 processor and 8GB of RAM, running Windows 7. LCNFQ has four feedforward neural networks for (21) and three feedforward neural networks for (19), each associated with an automaton state in Fig. 6.a and Fig. 6.b. We assume that the rover lands on a random safe place and has to find its way to satisfy the given property in the face of uncertainty. The learning discount factor γ is also set to be equal to 0.9.

Fig. 18 gives the results of learning for LTL formulae (21) and (19). At each state $s^{\otimes}$, the robot picks an action that yields highest $Q(s^{\otimes}, \cdot)$ and by doing so the robot is able to generate a control policy $Pol^{\otimes *}$ over the state space $\mathcal{S}^{\otimes}$. The control policy $Pol^{\otimes *}$ induces a policy Pol^{*} over the state space $\mathcal{S}$ and its performance is shown in Fig. 18.

Next, we investigate the episodic VQ algorithm as an alternative solution to LCNFQ. Three different resolutions ($\Delta = 0.4,\ 1.2,\ 2$ km) are used to see the

effect of the resolution on the quality of the generated policy. The results are presented in Table 1, where VQ with $\Delta = 2$ km fails to find a satisfying policy in both regions, due to the coarseness of the resulted discretisation. A coarse partitioning result in the RL not to be able to efficiently back-propagate the reward or the agent to be stuck in some random-action loop as sometimes the agent current cell is large enough that all actions have the same value. In Table 1, training time is the empirical time that is taken to train the algorithm and travel distance is the distance that agent traverses from initial state to final state. We show the generated policy for $\Delta = 1.2$ km in Fig. 19. Additionally, Fig. 21 depicts the resulted Voronoi discretisation after implementing the VQ algorithm. Note that with VQ only those parts of the state space that are relevant to satisfying the property are accurately partitioned.

Finally, we present the results of FVI method in Fig 20 for the LTL formulae (21) and (19). The FVI smoothing parameter is $h' = 0.18$ and the sampling time is $Z = 25$ for both regions where both are empirically adjusted to have the minimum possible value for FVI to generate satisfying policies. The number of basis points also is set to be 100, so the sample complexity of FVI is [7] equal to $100 \times Z \times |\mathcal{A}| \times (|\mathcal{Q}| - 1)$. Note that in Table 1, in terms of timing, FVI outperforms the other methods. However, we have to remember that FVI is an approximate DP algorithm, which inherently needs an approximation of the transition probabilities. Therefore, as we have seen in (18), for the set of basis points we need to sample the subsequent states. This reduces the FVI applicability as it is often not possible in practice.

Additionally, both FVI and episodic VQ need careful hyper-parameter tuning to generate a satisfying policy, i.e., h' and Z for FVI and Δ for VQ. The big merit of LCNFQ is that it does not need any external intervention. Further, as in Table 1, LCNFQ succeeds to efficiently generate a better policy compared to FVI and VQ. LCNFQ has less sample complexity while at the same time produces policies that are more reliable and also has better expected reward, i.e. higher probability of satisfying the given property.

5 Conclusions

In this paper we have proposed a framework to guide an RL agent, by expanding the agent domain knowledge about the environment by means of an LTL property. This additional knowledge, as we have observed in experiments, boosts the agent learning of the global optimal policy. Further we have shown that we can calculate the probability that is associated with the satisfaction of the LTL property that enables us to quantify the safety of the generated optimal policy at any given state.

[7] We do not sample the states in the product automaton that are associated to the accepting state of the automaton since when we reach the accepting state the property is satisfied and there is no need for further exploration. Hence, the last term is $(|\mathcal{Q}| - 1)$. However, if the property of interest produces an automaton that has multiple accepting states, then we need to sample those states as well.

We have argued that converting the LTL property to an LDBA results in a significantly smaller automaton than DRA alternatives, which increases the convergence rate of RL. In addition to the more succinct product MDP and faster convergence, our algorithm is easier to implement as opposed to standard methods that convert the LTL property to a DRA due to the simpler accepting conditions of LDBA. Much like the way we synchronised the states of LDBA with the states of the MDP, we have shown that synchronising a Kripke structure with the LDBA allows us to handle time-varying periodic environments on-the-fly. This particularly becomes important when we employed this synchronised LDBA-Kripke automaton as an infrastructure for the agent to transfer its learning over the dimension of time and to overcome the curse of dimensionality.

Last but not least, LCNFQ is the first RL algorithm that can generate policies in a continuous-state MDP that are safe with respect to an LTL formula. LCNFQ is model-free, meaning that the learning only depends on the sample experiences that the agent gathered by interacting and exploring the MDP. Further, the sample set can be small thanks to the generalisation that neural nets offer. The core engine in LCNFQ is very flexible and can be extended to the most recent developments in RL literature.

For future work we are currently looking into a multi-agent setup, in which a heterogeneous set of agents attempts to satisfy an LTL formula (or set thereof). Further, we would like to extend this approach to partially observable MDPs to limit the knowledge of the agent even more.

References

1. M. L. Puterman, *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
2. M. van Otterlo and M. Wiering, "Reinforcement learning and Markov decision processes," in *Reinforcement Learning*, pp. 3–42, Springer, 2012.
3. D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic Programming*, vol. 1. Athena Scientific, 1996.
4. D. Bertsekas, "Convergence of discretization procedures in dynamic programming," *IEEE Transactions on Automatic Control*, vol. 20, no. 3, pp. 415–419, 1975.
5. M. Prandini and J. Hu, "A stochastic approximation method for reachability computations," in *Stochastic Hybrid Systems*, pp. 107–139, Springer, 2006.
6. F. Dufour and T. Prieto-Rumeau, "Approximation of Markov decision processes with general state space," *Journal of Mathematical Analysis and Applications*, vol. 388, no. 2, pp. 1254–1267, 2012.
7. S. E. Z. Soudjani, C. Gevaerts, and A. Abate, "FAUST^2: Formal Abstractions of Uncountable-STate STochastic Processes," in *TACAS*, pp. 272–286, Springer, 2015.
8. M. S. Santos and J. Vigo-Aguiar, "Analysis of a numerical dynamic programming algorithm applied to economic models," *Econometrica*, pp. 409–426, 1998.
9. R. Munos and A. Moore, "Variable resolution discretization in optimal control," *Machine learning*, vol. 49, no. 2-3, pp. 291–323, 2002.
10. J. Stachurski, "Continuous state dynamic programming via nonexpansive approximation," *Computational Economics*, vol. 31, no. 2, pp. 141–160, 2008.
11. R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.
12. C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
13. R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.
14. S. Rahili and W. Ren, "Game theory control solution for sensor coverage problem in unknown environment," in *CDC*, pp. 1173–1178, IEEE, 2014.
15. H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *ACM Workshop on Networks*, pp. 50–56, ACM, 2016.
16. M. Hasanbeig and L. Pavel, "On synchronous binary log-linear learning and second order Q-learning," in *The 20th World Congress of the International Federation of Automatic Control (IFAC)*, IFAC, 2017.
17. D. Sadigh, E. S. Kim, S. Coogan, S. S. Sastry, and S. A. Seshia, "A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications," in *CDC*, pp. 1091–1096, IEEE, 2014.
18. P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," *NIPS*, vol. 19, p. 1, 2007.
19. Z. Zhou, X. Li, and R. N. Zare, "Optimizing chemical reactions with deep reinforcement learning," *ACS central science*, vol. 3, no. 12, pp. 1337–1344, 2017.
20. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
21. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.
22. K. Doya, "Reinforcement learning in continuous time and space," *Neural computation*, vol. 12, no. 1, pp. 219–245, 2000.
23. R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *NIPS*, pp. 1038–1044, 1996.
24. D. Ormoneit and Ś. Sen, "Kernel-based reinforcement learning," *Machine learning*, vol. 49, no. 2, pp. 161–178, 2002.

25. D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *JMLR*, vol. 6, no. Apr, pp. 503–556, 2005.
26. L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*, vol. 39. CRC press, 2010.
27. K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.
28. G. Tesauro, "TD-Gammon: A self-teaching Backgammon program," in *Applications of Neural Networks*, pp. 267–285, Springer, 1995.
29. V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *ICML*, pp. 1928–1937, 2016.
30. M. Riedmiller, "Neural fitted Q iteration-first experiences with a data efficient neural reinforcement learning method," in *ECML*, vol. 3720, pp. 317–328, Springer, 2005.
31. H. Van Hasselt and M. A. Wiering, "Reinforcement learning in continuous action spaces," in *ADPRL*, pp. 272–279, IEEE, 2007.
32. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
33. P. Abbeel, *Apprenticeship learning and reinforcement learning with application to robotic control*. Stanford University, 2008.
34. Y. Aytar, T. Pfaff, D. Budden, T. L. Paine, Z. Wang, and N. de Freitas, "Playing hard exploration games by watching youtube," *arXiv preprint arXiv:1805.11592*, 2018.
35. A. Billard, S. Calinon, R. Dillmann, and S. Schaal, "Robot programming by demonstration," in *Springer handbook of robotics*, pp. 1371–1394, Springer, 2008.
36. M. Večerík, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, "Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards," *arXiv preprint arXiv:1707.08817*, 2017.
37. T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, *et al.*, "Deep q-learning from demonstrations," *arXiv preprint arXiv:1704.03732*, 2017.
38. I.-A. Hosu and T. Rebedea, "Playing atari games with deep reinforcement learning and human checkpoint replay," *arXiv preprint arXiv:1607.05077*, 2016.
39. Z. Lipton, X. Li, J. Gao, L. Li, F. Ahmed, and L. Deng, "Bbq-networks: Efficient exploration in deep reinforcement learning for task-oriented dialogue systems," *arXiv preprint arXiv:1711.05715*, 2017.
40. T. Pohlen, B. Piot, T. Hester, M. G. Azar, D. Horgan, D. Budden, G. Barth-Maron, H. van Hasselt, J. Quan, M. Večerík, *et al.*, "Observe and look further: Achieving consistent performance on atari," *arXiv preprint arXiv:1805.11593*, 2018.
41. A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Overcoming exploration in reinforcement learning with demonstrations," *arXiv preprint arXiv:1709.10089*, 2017.
42. A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science*, pp. 46–57, IEEE, 1977.
43. A. P. Nikora and G. Balcom, "Automated identification of ltl patterns in natural language requirements," in *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pp. 185–194, IEEE, 2009.
44. R. Yan, C.-H. Cheng, and Y. Chai, "Formal consistency checking over specifications in natural languages," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1677–1682, EDA Consortium, 2015.
45. E. Gunter, "From natural language to linear temporal logic: Aspects of specifying embedded systems in ltl," in *Proceedings of the Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, 2003.
46. C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of Model Checking*. MIT press, 2008.
47. S. Safra, "On the complexity of omega-automata," in *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pp. 319–327, IEEE, 1988.

48. N. Piterman, "From nondeterministic Büchi and Streett automata to deterministic parity automata," in *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*, pp. 255–264, IEEE, 2006.
49. R. Alur and S. La Torre, "Deterministic generators and games for LTL fragments," *TOCL*, vol. 5, no. 1, pp. 1–25, 2004.
50. S. Sickert, J. Esparza, S. Jaax, and J. Křetínskỳ, "Limit-deterministic Büchi automata for linear temporal logic," in *CAV*, pp. 312–332, Springer, 2016.
51. I. Tkachev, A. Mereacre, J.-P. Katoen, and A. Abate, "Quantitative model-checking of controlled discrete-time Markov processes," *Information and Computation*, vol. 253, pp. 1–35, 2017.
52. E. M. Wolff, U. Topcu, and R. M. Murray, "Robust control of uncertain Markov decision processes with temporal logic specifications," in *CDC*, pp. 3372–3379, IEEE, 2012.
53. J. Fu and U. Topcu, "Probably approximately correct MDP learning and control with temporal logic constraints," in *Robotics: Science and Systems X*, 2014.
54. T. Brázdil, K. Chatterjee, M. Chmelík, V. Forejt, J. Křetínskỳ, M. Kwiatkowska, D. Parker, and M. Ujma, "Verification of Markov decision processes using learning algorithms," in *ATVA*, pp. 98–114, Springer, 2014.
55. M. Grześ, "Reward shaping in episodic reinforcement learning," in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pp. 565–573, International Foundation for Autonomous Agents and Multiagent Systems, 2017.
56. M. Kearns and S. Singh, "Near-optimal reinforcement learning in polynomial time," *Machine learning*, vol. 49, no. 2-3, pp. 209–232, 2002.
57. A. L. Strehl and M. L. Littman, "An analysis of model-based interval estimation for markov decision processes," *Journal of Computer and System Sciences*, vol. 74, no. 8, pp. 1309–1331, 2008.
58. K. Lesser and A. Abate, "Multiobjective optimal control with safety as a priority," *IEEE Transactions on Control Systems Technology*, vol. 26, no. 3, pp. 1015–1027, 2018.
59. M. Svorenova, I. Cerna, and C. Belta, "Optimal control of MDPs with temporal logic constraints," in *CDC*, pp. 3938–3943, IEEE, 2013.
60. A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, and J. H. Taankvist, "Uppaal stratego," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 206–211, Springer, 2015.
61. M. Wen, R. Ehlers, and U. Topcu, "Correct-by-synthesis reinforcement learning with temporal logic constraints," in *IROS*, pp. 4983–4990, IEEE, 2015.
62. S. Junges, N. Jansen, C. Dehnert, U. Topcu, and J.-P. Katoen, "Safety-constrained reinforcement learning for MDPs," in *TACAS*, pp. 130–146, Springer, 2016.
63. O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.
64. M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta, "Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees," in *ICRA*, pp. 3227–3232, IEEE, 2010.
65. S. Pathak, L. Pulina, and A. Tacchella, "Verification and repair of control policies for safe reinforcement learning," *Applied Intelligence*, pp. 1–23, 2017.
66. O. Andersson, F. Heintz, and P. Doherty, "Model-based reinforcement learning in continuous environments using real-time constrained optimization.," in *AAAI*, pp. 2497–2503, 2015.
67. A. Abate, M. Prandini, J. Lygeros, and S. Sastry, "Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems," *Automatica*, vol. 44, no. 11, pp. 2724–2734, 2008.
68. I. Tkachev, A. Mereacre, J.-P. Katoen, and A. Abate, "Quantitative automata-based controller synthesis for non-autonomous stochastic hybrid systems," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pp. 293–302, ACM, 2013.
69. S. Haesaert, S. E. Zadeh Soudjani, and A. Abate, "Verification of general markov decision processes by approximate similarity relations and policy refinement," *SIAM Journal on Control and Optimization*, vol. 55, no. 4, pp. 2333–2367, 2017.
70. X. Li, C.-I. Vasile, and C. Belta, "Reinforcement learning with temporal logic rewards," *arXiv preprint arXiv:1612.03471*, 2016.

71. M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," *arXiv preprint arXiv:1708.08611*, 2017.
72. A. L. Thomaz and C. Breazeal, "Teachable robots: Understanding human teaching behavior to build more effective robot learners," *Artificial Intelligence*, vol. 172, no. 6-7, pp. 716–737, 2008.
73. D. P. Bertsekas and S. Shreve, *Stochastic optimal control: the discrete-time case*. 2004.
74. R. Durrett, *Essentials of stochastic processes*, vol. 1. Springer, 1999.
75. R. Cavazos-Cadena, E. A. Feinberg, and R. Montes-De-Oca, "A note on the existence of optimal policies in total reward dynamic programs with compact action sets," *Mathematics of Operations Research*, vol. 25, no. 4, pp. 657–666, 2000.
76. K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
77. A. Cauchy, "Méthode générale pour la résolution des systemes déquations simultanées," *Comp. Rend. Sci. Paris*, vol. 25, no. 1847, pp. 536–538, 1847.
78. M. Riedmiller, "Concepts and facilities of a neural reinforcement learning control architecture for technical process control," *Neural computing & applications*, vol. 8, no. 4, pp. 323–338, 1999.
79. L.-H. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3/4, pp. 69–97, 1992.
80. V. Forejt, M. Kwiatkowska, and D. Parker, "Pareto curves for probabilistic model checking," in *ATVA*, pp. 317–332, Springer, 2012.
81. E. A. Feinberg and J. Fei, "An inequality for variances of the discounted rewards," *Journal of Applied Probability*, vol. 46, no. 4, pp. 1209–1212, 2009.
82. M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV*, pp. 585–591, Springer, 2011.
83. C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, "The probabilistic model checker storm," *arXiv preprint arXiv:1610.08713*, 2016.
84. M. Kearns and S. Singh, "Near-optimal reinforcement learning in polynomial time," *Machine learning*, vol. 49, no. 2-3, pp. 209–232, 2002.
85. D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.
86. M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in *Neural networks*, pp. 586–591, IEEE, 1993.
87. I. S. Lee and H. Y. Lau, "Adaptive state space partitioning for reinforcement learning," *Engineering applications of artificial intelligence*, vol. 17, no. 6, pp. 577–588, 2004.
88. G. Voronoi, "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs.," *Journal für die reine und angewandte Mathematik*, vol. 134, pp. 198–287, 1908.
89. G. J. Gordon, "Stable function approximation in dynamic programming," in *Machine Learning*, pp. 261–268, Elsevier, 1995.
90. O. Hernández-Lerma and J. B. Lasserre, *Further topics on discrete-time Markov control processes*, vol. 42. Springer Science & Business Media, 2012.
91. R. W. Shonkwiler and F. Mendivil, *Explorations in Monte Carlo Methods*. Springer Science & Business Media, 2009.
92. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.
93. A. S. McEwen, C. M. Dundas, S. S. Mattson, A. D. Toigo, L. Ojha, J. J. Wray, M. Chojnacki, S. Byrne, S. L. Murchie, and N. Thomas, "Recurring slope lineae in equatorial regions of Mars," *Nature Geoscience*, vol. 7, no. 1, p. 53, 2014.