Domain-independent reward machines for modular integration of planning and learning

Giuseppe De Giacomo[1*], **Marco Favorito**[1], **Luca Iocchi**[1], **Fabio Patrizi**[1]
[1]DIAG – Sapienza Univeristà di Roma, Italy
lastname@diag.uniroma1.it

Abstract

Integrating planning and learning components has many advantages in practical applications, as it allows for combining the different benefits of the two approaches: prediction of future states from planning with adaptivity to current situations from learning. However, a problem with is approach is that the two components should share a common representation of the information about the environment (e.g., states and actions). Previous work addresses this problem in the case where planning and learning are defined over different state variables, by defining a joint state space and a mapping between the two representations. In this paper, we present a method for integrating planning and reinforcement learning using a modular design where the two components can use their own representation formalism, without requiring an explicit mapping between them. More specifically, we introduce the concept of domain-independent reward machines, generated by a goal-oriented planning system and use them to drive a reinforcement learning agent to reach a goal state. Moreover, we show how to automatically generate and use sub task decomposition to speed up the reinforcement learning process.

Introduction

Reinforcement Learning (RL) (Sutton and Barto 2018) is a powerful tool for computing optimal behaviors of an agent, by collecting experience during the execution of some task and without requiring any knowledge about the environment's model. Many algorithms have been developed to explore the environment in an efficient way. Some model-based approaches aim at reconstructing the underlying dynamic system, typically a Markov Decision Process (MDP), during the learning phase. A more recent one also aims at extracting knowledge (using a symbolic formalism) from RL trials.

Hierarchical RL (HRL) is an extension of RL, where the problem is organized in a hierarchy of sub-problems, with the aim of speeding up the learning process. The use of factored MDPs, i.e., where states are modeled using state variables, in model-based RL allows for introducing a-priori knowledge expressed in a symbolic formalism.

[*]Contact Author

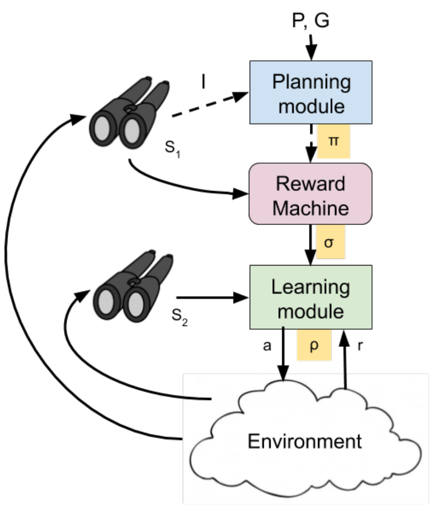

Figure 1: Architectural schema

In these approaches, the model is fully specified, and this includes mappings that relate the various hierarchical levels. However, HRL typically takes advantage of Knowledge Engineering (i.e., knowledge representation and reasoning tools) to speedup the learning process, rather than for specifying goals.

Very recent works have adopted logical specifications to express (temporal) goals. When temporal logics, such as LTL , are used, problems involving Non-Markovian rewards can easily be formalized (i.e., Non-Markovian Reward Decision Processes, NMRDP). Examples of such approaches include Reward Machines (Icarte et al. 2018) and Restraining Bolts (De Giacomo et al. 2019), which exploit finite-state machines as a way to specify RL agents' rewards.

When hierarchical structures are considered, several mechanisms to speed up the learning process can be used, such as options (Sutton, Precup, and Singh 1999), policy sketches (Andreas, Klein, and Levine 2017), etc. However, such previous approaches require additional modelling effort and, in hierarchical settings, also a mapping between the different representation layers. Automatic generation of sub tasks in HRL is still an open problem.

In this paper, we start from the work described in Icarte et al. 2018, where HRL, non-Markovian rewards, and task decomposition, are combined into the Reward Machine framework. Similarly to that, we also consider two separate representation layers, one for the goal and one for the learning agent, but do not require any mapping between them. Specifically, we consider the setting depicted in Fig. 1, where three distinct modules are used to learn an optimal behavior (policy) over the environment:

- a *planning module*, which performs high-level *offline* reasoning and generates a plan π, used to guide, and speedup, the learning process;
- a *reward machine*, which is, essentially, a runtime monitor that observes the environment from the high-level perspective and returns a signal σ whenever the input plan π advances one step towards the goal;
- a *learning module*, which observes the environment from a low-level perspective, interacting with it through actions and possible rewards, and receiving the signals σ from the reward machine.

The two perspectives of the environment are based on the use of two independent sets of sensors, S_1 and S_2, each extracting its own state variables (or features) of the environment.

Notice that the planning module operates offline and state variables observed by S_1 are used to detect the initial state I (dashed line in Fig. 1) to start the plan generation process. At plan execution time, the state variables observed by the reward machine through S_1 take values consistently with the observations of the learning agent through S_2.

As said, this setting is very similar to that of (Icarte et al. 2018), with the crucial difference that we do not require an explicit mapping between the two representation layers and thus the reward machine does not need to be an input for the RL algorithm but can be kept as a separate component. In other words, in our work the RL agent does not know the reward machine, but it just receives domain-independent signals from it. Of course, an implicit mapping exists, as the two representations derive from observations of the same environment, however it is not necessary to make this mapping explicit to the RL agent and thus the proposed approach does not require additional modeling efforts. Nonetheless, in some cases, in order to ensure such implicit relation, we may require the set of sensors to be synchronized (i.e., the components should share a common clock).

Since the learning component does not require a representation of the domain model specified at the planning level and, consequently, the reward machine can produce only signals not depending on the state representations used by the planning and learning modules, both components are referred to as *domain-independent*. Such *domain-independent* components can thus be designed and implemented separately, allowing the system to be highly modular. For example, planner modules with different representations of states and actions can be interchanged without requiring any modification of the implementation of the learning module; the same reward machine can be applied to different learning agents with different representations of states and actions without requiring any modification on the agent.

In this paper, we show that, although the planning and learning modules are loosely linked with domain-independent signals only, they can still cooperate to reach a common goal, as they observe and act (directly or indirectly) on the same environment. We also discuss how such mechanism allows for an easy way to automatically define and exploit sub task decomposition, to speedup the learning process and, finally, report on experimental results. The original contributions of this paper are the following.

Firstly, we propose an approach for the integration of a planning component (or more in general a reasoning system) with a RL agent, in a setting where the components use different representation formalisms (for example, different state variables and different actions), without requiring an explicit mapping. This simplifies the previous approach based on reward machines (Icarte et al. 2018) as reducing the required modeling effort and, more importantly, broadens the range of applicability of the approach, to those situations where a mapping is not simple to define or not possible at all.

Secondly, we present a mechanism to automatically generate sub task decomposition, that can be used to speedup the learning process, by extending previous work on restraining bolts (De Giacomo et al. 2019), demonstrating faster convergence when sub task decompositions are considered.

Related work

A general approach for integrating planning and learning is given by model-based RL where the goal is to reconstruct the model of the environment while learning. Dyna (Sutton 1990) and R-Max (Brafman and Tennenholtz 2002) are example of such methods in which the learning experience is used to build a model of the environment and such a model is used to generate policies followed during the learning process. The use of inaccurate models and few real trials to speed-up learning is also presented in (Abbeel, Quigley, and Ng 2006). In all these works the behavior of the agent is guided by a reward function that is assumed to be available and sampled during agent execution.

The use of a planner to drive the RL process is shown for example in (Grzes and Kudenko 2008; Efthymiadis and Kudenko 2014) where it is used to define a reward shaping functions to drive the agent along the plan, and in (Leonetti, Iocchi, and Stone 2016) to constrain the exploration space of the agent, by defining partial policies in which each state is associated with a set of possible actions that will be considered during the exploration phase of RL. In these works, the planning domain is defined on the same state representation that is used by the RL algorithm and it is thus deeply linked to it.

Hierarchical Reinforcement Learning (HRL) and options (Sutton, Precup, and Singh 1999) are also commonly used to speed-up the learning process. While, in general, the models used at the layers of the hierarchical architecture and the options are manually defined, there have been some approaches to generate them automatically with a planning component, such as (Grounds and Kudenko 2007; Yang et al. 2018). Also in these cases, either the planning and the learning components share the same representation of the states, or an explicit mapping between these representations is required.

Finally, more general approaches to drive the RL process of an agent, considering also temporal goals and non-Markovian rewards are reward machines (Icarte et al. 2018) and restraining bolts (De Giacomo et al. 2019). The use of an automated planner to generate controllers for reward machines is also presented in (Leon Illanes et al. 2019), but again an explicit mapping between the representations used by the planner component and the learning agent is required. While the restraining bolts described in (De Giacomo et al. 2019) use a different representation with respect to the one used by the agent, without requiring an explicit mapping between such representations. However, this work does not describe the generation of the bolt that is assumed to be given.

The method described in this paper combines the advantages of previous works by defining a framework for automatic generation of a reward machine using model-based and goal-oriented planning, in order to drive the RL agent to learn a policy following the desired plan, thus achieving the desired goal. We solve this problem in the setting in which the planning component and the learning component use different representations of states and actions involved in the task and a mapping between such representation is not required. Such a reward machine, whose states must not be mapped to the states of the learning agent, is called in this paper *domain-independent* to emphasize its modularity. Indeed, as shown later, a *domain-independent reward machine* communicates with the RL agent through domain-independent messages (i.e., messages not expressed in terms of the representation used to formalize the planning problem). Consequently, a domain-independent reward machine can be placed on any RL agent without requiring additional specifications or modifications of such an agent.

Problem formulation

The hierarchical architecture we consider is reported in Fig. 1. A planning module generates a plan π, given a domain model and a goal G. The plan is then used to generate a *reward machine* (see below), to guide the learning process of a learning agent, by providing suitable signals σ during the learning process. We next introduce the basic components.

Planning module. A *(deterministic) planning domain* is a tuple $\mathcal{P} = \langle V, A_D \rangle$, with V a finite set of boolean state variables and A_D a set of action descriptions (e.g., in terms of preconditions and effects over V). A state W of $\mathcal{P}$ is a (total) assignment to the variables in V, represented as a set $W \in 2^V$, s.t. $W \in V$ is true iff $W \in q$. Actions are intended to be executed in a state W and lead to exactly one (in the deterministic setting) successor state W'. Given a planning domain $\mathcal{P}$, an initial state I and a goal G (expressed as a formula over V), an automatic planner generates, if any, a *plan* $\pi = \alpha_0, \alpha_1, \ldots, \alpha_n$, i.e., a finite sequence of actions that, when executed from the initial state I, takes $\mathcal{P}$ to a state satisfying G. In our setting, the values stored in variables from V are consistent with the values returned by the sensors in S_1, which observe the environment. In this paper we consider offline planning. However, the proposed approach is not limited to this case and can be adapted to online planning and replanning. In this case, the RL agent will adapt to new plans, and thus new reward machines, through experience (this requires that changes in plans occur much less frequently than RL agent's action executions).

Reinforcement learning module. A *Markov Decision Process* (MDP) is a tuple $\mathcal{M} = \langle S, A, Tr, R \rangle$ containing: a set S of states; a set A of actions; a transition function $Tr : P(s'|s, a)$ returning, for every state s and action a, a probability distribution over the next state s'; and a reward function $R : S \times A \times S \to \Re$ that specifies the reward (a real value) received by the agent when transitioning from state s to state s' by applying action a. The states S of the MDP are observed with a set of sensors S_2 that are in general different from those in S_1, used at the planning level. A solution to an MDP is a function $\rho : S \mapsto A$, called *policy*, assigning an action to each state. The *value* of a policy ρ at state s, denoted $v^\rho(s)$, is the expected sum of the rewards obtained when starting at state s and selecting actions based on ρ (possibly discounted by a factor γ, with $0 \leq \gamma \leq 1$). Reinforcement learning agents are designed to find optimal polices over MDPs.

Reward machine. In this paper, planning and learning are integrated through the definition of a *reward machine* (Icarte et al. 2018) over the state variables V. This machine acts as a runtime monitor observing the running environment from the same (high-level) perspective as the planning module, and sending suitable signals σ to the learning module, depending on the advancement of the environment state wrt the plan π returned by the planning module. Formally, a *reward machine* is a tuple $\mathcal{RM} = \langle Q, q_0, \delta_q, \delta_r \rangle$, where: Q is a finite set of states; $q_0 \in Q$ is an initial state; $\delta_q : Q \times 2^V \mapsto Q$ is a state-transition function; and $\delta_r : Q \times Q \mapsto \Re$ is a reward-transition function. The reward machine is automatically generated from a plan π, as described in (Leon Illanes et al. 2019). However, in contrast with that work, we do not require: i) to define a (new) joint space state including both V from the planning module and S from the learning module, or ii) to explicitly relate S and V. Consequently, our $\mathcal{RM}$ is defined by only considering elements at the planning level (i.e., only using the state variables V captured by sensors S_1), without relating them to states S and actions A used by the learning agent. More specifically, $Q_\pi \subseteq 2^V$ contains all the states traversed during the execution of plan π, $q_0 = I$ is the initial state, δ_q is defined over transitions of state variables in V and δ_r is associated only to transitions in Q.

The problem In this paper we address the problem of learning an optimal policy over the MDP $\mathcal{M}$, using the architecture described above, in particular without requiring any explicit mapping to connect the various layers. The proposed solution defines information σ to be shared between these two modules, that is domain-independent, i.e., not based on V or S.

Solution

The solution is based on the creation of a reward machine that controls the learning process of the RL agent by using only domain-independent signals. Below, we describe the steps to generate the machine and the use of options associated with it.

Reward machine generation

The reward machine is automatically obtained by first deriving a transition graph from the plan π generated by the planner and then by associating reward values with state transitions. Specifically, a plan π can be transformed into a transition graph $\mathcal{T}_\pi = \langle Q_\pi, q_{0\pi}, E_\pi \rangle$ where: $Q_\pi \subseteq 2^V$ is the set of states traversed during the execution of π over the planning domain $\mathcal{P}$; $q_{0_\pi} = I \in Q_\pi$ is the initial state of the planning problem; and $E_\pi \subseteq Q_\pi \times Q_\pi$ is the set of edges (i.e., the actions occurring in π) connecting two states in Q_π, according to the execution of π. By exploiting the reasoning capabilities of the planning system, it is possible to associate each state $q \in Q_\pi$ with a formula $\phi(q)$ over state variables V denoting the set of states that can be reached after the execution of the plan up to that state. Since we focus on classical planners generating sequential plans, the transition graph $\mathcal{T}_\pi$ is a linear graph with initial node I, one edge for each action α_i, and a final node where the goal G is satisfied.

A reward machine $\mathcal{RM}_\pi$ can now be derived from the transition graph $\mathcal{T}_\pi$ by just adding a mechanism to associate rewards with state transitions. A straightforward implementation consists in assigning a high positive rewards to the transitions reaching a goal state and zero to the other transitions. Thus, $\mathcal{RM}_\pi = \langle Q_\pi, q_{0_\pi}, \delta_q, \delta_r \rangle$, where $\delta_q(q, \phi(q')) = q'$ iff $(q, q') \in E_\pi$ and $\phi(q')$ denotes the set of states denoted by the formula $\phi(q')$ (i.e., states reached after the execution of the plan up to state q'), and $\delta_r(q, q') > 0$ if $q' \in G$ is a goal state, 0 otherwise. In practice, forms of reward shaping applied to the reward machine can help in speeding up the learning process (Camacho et al. 2019).

Use of the reward machine for RL

The reward machine continuously monitors the evolution of the plan and reports to the underlying RL agent the information necessary to guarantee that the RL agent would converge to a policy that will reach a goal state. To this end, the $\mathcal{RM}$ checks occurrence of a transition $\delta_q(q_t, \phi(q_{t+1})) = q_{t+1}$ in the current state q_t. When $\phi(q_{t+1})$ becomes true (as observed through sensors S_1), then a state transition is detected and communicated to the RL agent. Upon detecting a state-transition, the $\mathcal{RM}$ performs the following operations: 1) updates current and past states: $q_{t-1} \leftarrow q_t$, $q_t \leftarrow q_{t+1}$, 2) sends signal $\sigma_t = \langle \hat{q}_t, r_t \rangle$ to the RL agent, where: $\hat{q}_t$ is an encoding of the current machine's state q_t, and $r_t = \delta_r(q_{t-1}, q_t)$ is the reward value associated with the current machine's transition.

Importantly, observe that the encoding $\hat{q}_t$ can be any, as long as not expressed in terms of V, but in a domain-independent way. For example, it can be an integer corresponding to the index of q_t in some enumeration of Q_π.

In order to accept such information, the RL agent must be extended with a single variable to represent the encoding of the state of the reward machine (for example, an integer variable) and must take into account additional rewards coming from the reward machine. Therefore, the RL agent will act on a new MDP $\mathcal{M}' = \langle S \times \hat{Q}, A, Tr', R' \rangle$, where $S \times \hat{Q}$ is the extended space state including the encoding $\hat{Q}$ of the state of the reward machine (e.g., an integer value), Tr' and R' are the extended transition and reward functions that are unknown to the agent and thus we do not need to specify them. Notice that R' is extended by summing rewards r_t coming from the reward machine, in addition to the rewards coming from the environment. In order to guarantee reaching plan goals, we require rewards coming from the reward machine to be (significantly) higher than the rewards coming from the environment. When achieving the planning goal is the only objective of the agent, we can set to zero all the rewards coming from the environment.

We observe that the notion of $\mathcal{RM}$s is essentially analogous to that of Restraining Bolts (RB) proposed in (De Giacomo et al. 2019), i.e., runtime monitors offering rewards when favorable state transitions occur. In fact, the whole setting we consider here is analogous to that of (De Giacomo et al. 2019), thus we can take advantage of the results reported there. In particular, Th. 6 states that if the RL agent can accept rewards from the RB and can keep track of the RB current state, then any RL algorithm is successful in making the agent learn an optimal policy that enforces the RB (i.e., that achieves a goal state, in our case). Since, as it can be easily seen, the MDP $\mathcal{M}'$ defind above captures exactly this situation (the reward includes the $\mathcal{RM}$'s and the agent state is extended to accommodate a representation of the current $\mathcal{RM}$ state), it turns out that we can learn an optimal policy by operating on $\mathcal{M}'$.

Automatic sub task decomposition

In this paper we follow the sub-task decomposition induced by the reward machine, as proposed in (Icarte et al. 2018), by associating different q functions to the states of the $\mathcal{RM}$. However, as a difference with QRM algorithm proposed in (Icarte et al. 2018), in this work we focus on single task scenarios and we use an on-policy method. Possible use of off-policy methods to learn in parallel multiple reward machines associated to different tasks is left as future work.

Moreover, in our framework, we exploit domain-independent specifications of the $\mathcal{RM}$ and of the RL agent to implement a mechanism to enable/disable exploration for each sub-task, in order to speed-up convergence. This mechanism is similar to *options* (Sutton, Precup, and Singh 1999) or other techniques for learning sub-tasks in hierarchical RL (Hengst 2010)

In our implementation, sub-tasks are defined as pairs of transition (q_{t-1}, q_t) of the $\mathcal{RM}$ and the RL agent can detect start and end of a sub-task from the signals σ emitted by the $\mathcal{RM}$. Therefore, each signal received by the RL agent from the $\mathcal{RM}$ indicates the end of the previous sub-task and the start of a new one. At this moment, the RL agent can decide to enable/disable exploration for this sub-task until the next signal. When exploration is disabled, the agent actually exploits the current policy to achieve the current sub task, while when exploration is enables, the agent learns how to improve its policy for the current sub task. This mechanism allows for speeding up the learning process by avoiding exploration steps for sub tasks for which the current policy is good enough (or optimal).

We can define different criteria for deciding when to enable/disable exploration for sub tasks, ranging from ϵ-greedy to more informed probabilistic selections. Possible criteria include: A) a constant ϵ-greedy approach, B) a variable ϵ-greedy approach considering the number of visits of the transition (q_{t-1}, q_t), C) a probabilistic choice based on percentage of success in the transition (q_{t-1}, q_t). In this paper, we focus on evaluation of criterion C, since we consider environments with failure states in the reward machine that prevent the agent to proceed towards the goal and thus make the percentage of success in a transition a relevant choice.

The full procedure for extending a RL algorithm to control sub task exploration is described through the following snippets of algorithms. Here we refer to an on-policy method where the decision of enabling/disabling exploration for a sub task is applied to the policy being learned.

We make use of a variable $exploration\_ON$ that denotes whether exploration (i.e., choosing actions not only according to the best values of the current policy) is enabled for the current task or not. This choice is kept for the entire execution of the current sub task. On receiving a message from the reward machine, the agent chooses how to operate for the next sub task.

```
**Variable** $exploration\_ON$ // exploration is enabled

**Function** $choose\_action()$:
if $exploration\_ON$ then
    $\epsilon$-greedy choice
else
    choose_best_action

**Function** $on\_receive(\sigma_t)$:
$exploration\_ON$ = subtask_expl_criterion()

**Function** subtask_expl_criterion():
$p = Success(q_{t-1}, q_t)/Visits(q_{t-1}, q_t)$
return $random\_value(0, 1) > p$
```

Experimental results

To show the effectiveness of the proposed method we performed some experiments in three settings already used in previous works in RL: Breakout, Sapientino and Minecraft. These experimental scenarios have been used in (De Giacomo et al. 2019) to evaluate restraining bolts, while here we consider a more general setting in which reward machines are generated by automated planning procedures. The tasks to be learned (see (De Giacomo et al. 2019) for details) require the agents to perform sequences of actions. The problems can be easily modelled with a planning language and corresponding plans can be generated by suitable planners. More specifically, in Breakout we consider the goal of breaking the columns of bricks in a given order (e.g., from left to right), so the high-level plan is a sequence of actions $break\_column\_i$. In Sapientino, a particular sequence of actions (bip) must be executed according to the colors of the cells in which the little robot is, so the plan is a proper sequence of $goto\_xy$ and bip actions. Finally, the goal of Minecraft is to achieve 10 sub-goals by combining proper sequences of actions that must be executed in the environment at proper locations.

Notice that in these examples, state variables observed at the different layers (reward machine and RL agent) are disjoint (more details) below, but we do not need to modify the state representations of the RL agent according to the specific reward machine. In order to use existing algorithms (e.g., QRM (Icarte et al. 2018)) in these domains, additional modelling and modifications are necessary: 1) extend the state representation of the RL agent by considering the state variables used in the reward machine, 2) provide the labelling function bewteen states of the RL agent and state variables of the $\mathcal{RM}$.

More specifically, in Breakout the state representation contains the paddle position, ball position and velocity, but not the configuration of the bricks, while available actions are just to move left or right. In Sapientino and Minecraft, the agent only knows its position in the grid (but not the color of the cells for Sapientino or the presence of resources or tools for Minecraft) and actions are one-step movements on the grid.

The plots in Figure 2 show learning performance in the three domains.

In the top row, we see an example of different reward machines applied to the same agent. The Breakout agent can move in the environment to intercept a ball and break the bricks in the environment. The two reward machines differ in the plan they represent: in the first case (plot in the left), the plan is to break four columns from left to right, in the second case (plot in the right), the plan is to break the columns from right to left.

In the middle row we show an example of using the same reward machine on two different agents: the reward machine drives the agents to learn a policy in which cells of the grid are visited in a specified order, the two Sapientino agents differ in the state representations and actions available. In particular, the first agent (plot on the left) is an omni-directional robot moving in the four cardinal directions, while the second agent (plot on the right) is a differential drive robot moving forward/backward and turning left/right. The second agent includes also orientation as state variable. As shown in the figure, both agents are able to learn a policy according to the same reward machine, although with different low-level capabilities.

Finally, the bottom row shows the same reward machine applied to different Minecraft agents: omni-directional and differential drive. The state representation of the RL agent for Minecraft is exactly the same as the one used in Sapientino, while the set of actions are different in the two cases.

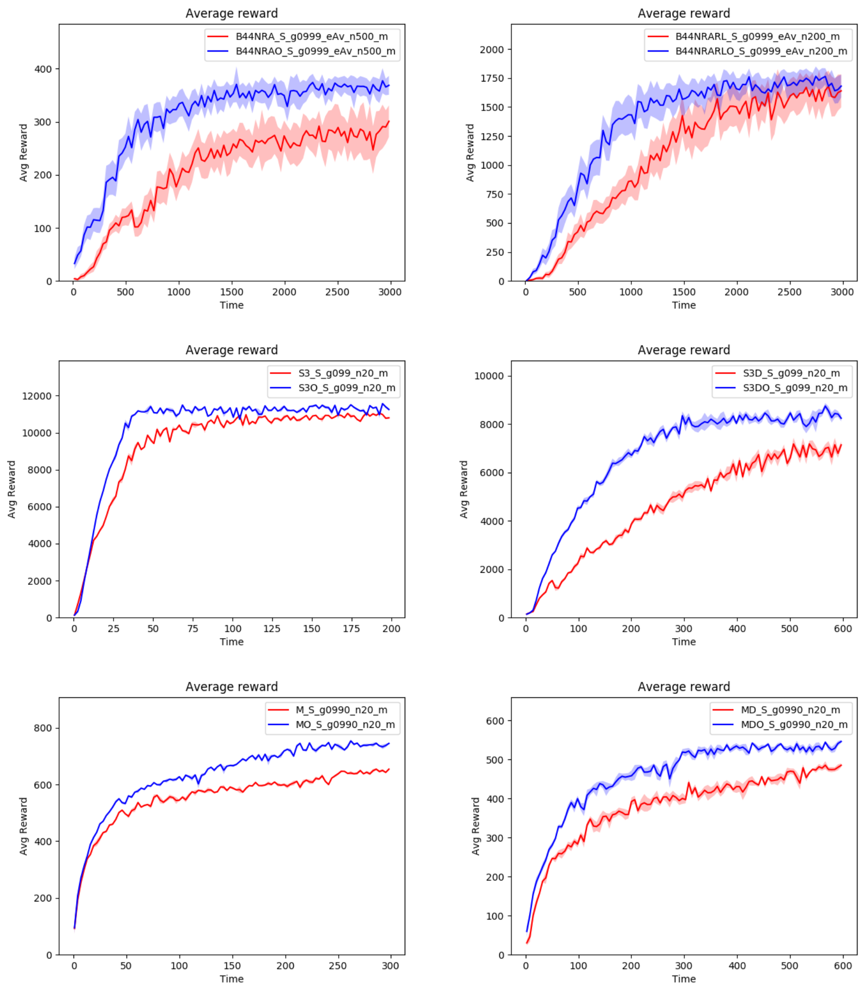

Figure 2: Average reward over experimental time. Breakout (top), Sapientino (middle), Minecraft (bottom), with sub task decomposition (blue), without sub task decomposition (red).

Notice that an agent with the same state representation can learn Sapientino or Minecraft tasks only based on information received by the reward machine without requiring to change its state representations.

In all the situations, the RL agent learns a policy that achieve the goal as specified by the planning module. Moreover, learning is improved when using automatic sub task decomposition (blue curves) with respect to the standard RL algorithm (red curves).

These experimental results thus show convergence to a policy reaching the goal and improved performance when using sub task decomposition. Moreover, the results have been obtained in a modular way, as results reported in left and right plots have been obtained by just composing different instances of reward machine and RL agent, without changing their internal representations.

Finally, consider that, when explicit mapping is required, any combinatin of $\mathcal{RM}$ and RL agent requires a specific modeling effort. Thus, to execute the 6 experiments reported above, a total of 12 (6 $\mathcal{RM}$ + 6 RL) components must be devised. Our approach, instead, allows for re-using and combining existing components, without any additional modeling effort. Specifically, wrt the 6 experiments above, we have defined only 3 RL agents (1 for Breakout and 2 for both Sapientino and Minecraft) and 4 $\mathcal{RM}$ (2 for Breakout and 1 for each other problem), for a total of 7 components.

Conclusion

Integration of planning and learning can benefit from modularity and separation of design and implementation of the relative components. In this paper, we have shown that state and action representations of the two layers can be kept completely separated and only domain-independent signals are needed to ensure to drive the learning process through a desired plan to reach a given goal.

Future work includes extension of the formalism to more complex forms of plans, such a partial order plans, hierarchical task networks, conditional plans, and plans represented through Petri nets that would allow to generate compact reward machines for compex tasks.

We believe that design and development of modular planning and learning components will be convenient in many application domains, making the integration of planning and learning easier and more effective.

Acknowledgments

This work is partially supported by the ERC Advanced Grant WhiteMech (No. 834228), by the EU ICT-48 2020 project TAILOR (No. 952215), and in part by Sapienza University of Rome through the Project DRAPE: Data-awaRe Automatic Process Execution.

References

Abbeel, P.; Quigley, M.; and Ng, A. Y. 2006. Using inaccurate models in reinforcement learning. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, 1–8. doi:10.1145/1143844.1143845. URL https://doi.org/10.1145/1143844.1143845.

Andreas, J.; Klein, D.; and Levine, S. 2017. Modular Multitask Reinforcement Learning with Policy Sketches. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, 166–175. PMLR.

Brafman, R. I.; and Tennenholtz, M. 2002. R-MAX - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *J. Mach. Learn. Res.* 3: 213–231. URL http://jmlr.org/papers/v3/brafman02a.html.

Camacho, A.; Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2019. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 6065–6073. doi:10.24963/ijcai.2019/840. URL https://doi.org/10.24963/ijcai.2019/840.

De Giacomo, G.; Iocchi, L.; Favorito, M.; and Patrizi, F. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019.*, 128–136.

Efthymiadis, K.; and Kudenko, D. 2014. A comparison of plan-based and abstract MDP reward shaping. *Connect. Sci.* 26(1): 85–99. doi:10.1080/09540091.2014.885283. URL https://doi.org/10.1080/09540091.2014.885283,https://www.tandfonline.com/doi/full/10.1080/09540091.2014.885283.

Grounds, M. J.; and Kudenko, D. 2007. Combining Reinforcement Learning with Symbolic Planning. In *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning, 5th, 6th, and 7th European Symposium, ALAMAS 2005-2007 on Adaptive and Learning Agents and Multi-Agent Systems, Revised Selected Papers*, 75–86. doi:10.1007/978-3-540-77949-0_6. URL https://doi.org/10.1007/978-3-540-77949-0_6.

Grzes, M.; and Kudenko, D. 2008. Plan-based reward shaping for reinforcement learning. In *Proc. of the 4th International IEEE Conference on Intelligent Systems*, 10–22.

Hengst, B. 2010. *Hierarchical Reinforcement Learning*, 495–502. Boston, MA: Springer US. ISBN 978-0-387-30164-8. doi:10.1007/978-0-387-30164-8_363. URL https://doi.org/10.1007/978-0-387-30164-8_363.

Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2018. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 2112–2121. URL http://proceedings.mlr.press/v80/icarte18a.html, http://proceedings.mlr.press/v80/icarte18a/icarte18a.pdf.

Leon Illanes, L.; Yan, X.; Icarte, R.; and McIlraith, S. 2019. Symbolic Planning and Model-Free Reinforcement Learning: Training Taskable Agents. In *Proc. of 4th Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*. URL http://www.cs.toronto.edu/~lillanes/papers/IllanesYTM-rldm2019-symbolic.pdf.

Leonetti, M.; Iocchi, L.; and Stone, P. 2016. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence* 241: 103–130. doi:10.1016/j.artint.2016.07.004. URL https://doi.org/10.1016/j.artint.2016.07.004.

Sutton, R. S. 1990. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Machine Learning, Proceedings of the Seventh International Conference on Machine Learning, Austin, Texas, USA, June 21-23, 1990*, 216–224. doi:10.1016/b978-1-55860-141-3.50030-4. URL https://doi.org/10.1016/b978-1-55860-141-3.50030-4.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, second edition. URL http://incompleteideas.net/book/the-book-2nd.html.

Sutton, R. S.; Precup, D.; and Singh, S. P. 1999. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artif. Intell.* 112(1-2): 181–211. doi:10.1016/S0004-3702(99)00052-1. URL https://doi.org/10.1016/S0004-3702(99)00052-1.

Yang, F.; Lyu, D.; Liu, B.; and Gustafson, S. 2018. PEORL: Integrating Symbolic Planning and Hierarchical Reinforcement Learning for Robust Decision-Making. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 4860–4866. doi:10.24963/ijcai.2018/675. URL https://doi.org/10.24963/ijcai.2018/675.