A Synthesis of Automated Planning and Reinforcement Learning for Efficient, Robust Decision-Making

Matteo Leonetti[a,c,*], Luca Iocchi[b], Peter Stone[a]

[a] *Department of Computer Science, The University of Texas at Austin, 2317 Speedway, Stop D9500, Austin, TX 78712, USA*
[b] *Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Via Ariosto 25, 00185 Rome, Italy*
[c] *School of Computing, University of Leeds, LS2 9JT Leeds, UK*

Abstract

Automated planning and reinforcement learning are characterized by complementary views on decision making: the former relies on previous knowledge and computation, while the latter on interaction with the world, and experience. Planning allows robots to carry out different tasks in the same domain, without the need to acquire knowledge about each one of them, but relies strongly on the accuracy of the model. Reinforcement learning, on the other hand, does not require previous knowledge, and allows robots to robustly adapt to the environment, but often necessitates an infeasible amount of experience. We present Domain Approximation for Reinforcement LearnING (DARLING), a method that takes advantage of planning to constrain the behavior of the agent to *reasonable* choices, and of reinforcement learning to adapt to the environment, and increase the reliability of the decision making process. We demonstrate the effectiveness of the proposed method on a service robot, carrying out a variety of tasks in an office building. We find that when the robot makes decisions by planning alone on a given model it often fails, and when it makes decisions by reinforcement learning alone it often cannot complete its tasks in a reasonable amount of time. When employing DARLING, even when seeded with the same model that was used for planning alone, however, the robot can quickly learn a behavior to carry out all the tasks, improves over time, and adapts to the environment as it changes.

Keywords: Automated Planning, Reinforcement Learning, Autonomous Robot, Robot Learning, Answer Set Programming

*Corresponding author
Email addresses: matteo@cs.utexas.edu (Matteo Leonetti), iocchi@dis.uniroma1.it (Luca Iocchi), pstone@cs.utexas.edu (Peter Stone)

1. Introduction

A great deal of work has been carried out in automated planning, and deliberation in general, while the application of such work to autonomous agents is much less developed [13]. The deployment of automated planning, especially in robotic tasks, faces many challenges, mostly due to high levels of uncertainty, which make completing a plan a significantly more difficult task than computing one. While low-level planning, such as navigation or motion planning, is gaining momentum in robotics, high-level planning is often avoided altogether. Behaviors are, instead, engineered by programming them directly.

The reason why plans are brittle can ultimately be attributed to imperfections in the models: relevant details overlooked, dynamics incorrectly represented, or assumptions violated. Nonetheless, making decisions in domains of any interest unavoidably involves abstraction and approximation, causing imperfect models to be widespread.

Execution Monitoring [24] and continual on-line planning [5] deal with unreliable models, but being able to recognize and react to failures might not be enough: an intelligent agent should learn from its mistakes and avoid them as much as possible in the future. The Reinforcement Learning (RL) [31] paradigm suits perfectly to this scenario, since it is based on trial and error, and the agent can have little knowledge about the domain before execution. Reinforcement Learning on its own, however, without prior knowledge of the environment, requires an enormous amount of experience. Applying RL methods directly is often infeasible in many practical cases, especially involving physical systems such as robots.

We propose to overcome the brittleness of the plans computed on a model through reinforcement learning in the environment, and to restrict the exploration of the environment through automated reasoning on the model. The resulting approach, Domain Approximation for Reinforcement LearnING (DARLING), exploits the synergy of the two methods allowing the agent, on the one hand, to relax the requirements on the planner, which can work on a simplified, abstract, representation of the domain. On the other hand, it allows the agent to take advantage of previous knowledge, for reducing the experience required by reinforcement learning. The automated reasoner provides a *rational* way to constrain the exploration and reduce the search space, while reinforcement learning eases the requirements on the accuracy of the model, which does not need to incorporate transition probabilities and action costs, even if the agent is in fact acting in a stochastic environment.

The main idea behind this work was introduced by Leonetti et al. [16] in the context of Hierarchical Reinforcement Learning, where a finite-state controller was induced by reasoning in Linear Temporal Logics (LTL), and used to constrain the exploration during a subsequent reinforcement learning phase. This article contributes a new formulation in terms of partial policies, an implementation of the planning phase through Answer Set Programming (ASP) instead of model-checking on LTL, and a thorough experimental validation conducted in several new domains. The use of ASP allowed us to scale the applicabil-

ity of DARLING to real-world domains. In particular we deployed it to our autonomous mobile robots, carrying out and learning several tasks in a real-world office environment (the Gates-Dell Complex at the University of Texas at Austin).

2. Background and Notation

This work leverages both reinforcement learning in Markov Decision Processes and automated reasoning in Answer Set Programming. In this section we introduce these two formalisms, along with the notation we will use in the rest of the article.

2.1. Markov Decision Processes

A Markov Decision Process is a tuple $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$ where:

- $\mathcal{S}$ is a set of *states*
- $\mathcal{A}$ is a set of *actions*. If not differently specified, every action is available in every state. When that is not the case, we denote with $\mathcal{A}(s)$ the set of actions available in state $s \in \mathcal{S}$.
- $f : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition function. The function $f(s, a, s') = Pr(S_{t+1} = s' | S_t = s, A_t = a)$ is the probability that the current state changes from s to s' by executing action a. If $f(s, a, s') = \{0, 1\}$ the system is said to be *deterministic*, otherwise it is *stochastic*.
- $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \to [0, 1]$ is the reward function. The function $r(s, a, s', g) = Pr(R_{t+1} = g | S_t = s, A_t = a, S_{t+1} = s')$ is the probability of getting a reward g for being in state s, executing action a, and reaching state s'. The reward is said to be *deterministic* if $r(s, a, s', g) = \{0, 1\}\ \forall s, s' \in \mathcal{S}$, $a \in \mathcal{A}$, and $g \in \mathcal{R}$. If the reward is negative, we will also refer to it as a *cost*.

We consider the system at discrete time steps. Let $t \in \mathbb{N}$ be the current time, and S_t be the state at time t. The agent interacts with the environment by choosing an action A_t and perceiving the next state S_{t+1}, such that:

$$S_{t+1} \sim f(S_t, A_t, \cdot) = Pr(S_{t+1} = s' | S_t = s, A_t = a),\ s, s' \in \mathcal{S} \text{ and } a \in \mathcal{A}$$

It also receives a reward R_{t+1}:

$$R_{t+1} \sim r(S_t, A_t, S_{t+1}, \cdot)$$

If a state is never left, after it is entered for the first time, it is said to be a *terminal* or an *absorbing* state. If s is a terminal state then $S_{t+1} = s$ holds almost surely given that $S_t = s$. If an MDP has a terminal state it is said to be *episodic*.

The behavior of the agent is represented as a function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ called a *(stationary) policy*, where $\pi(s, a) = Pr(A_t = a|S_t = s)$ is the probability of selecting action a in state s at time t. If $\pi(s, a) = \{0, 1\}\ \forall\, s \in \mathcal{S}$ and $a \in \mathcal{A}$ the policy is *deterministic*, in which case we will denote the action chosen by the policy as $a = \pi(s)$. A policy π and an initial state s_0 determine a probability distribution over the possible sequences $(\langle S_t, A_t, R_{t+1}\rangle,\ t \geq 0)$. Given such a sequence, we define the *cumulative discounted reward* as:

$$G = \sum_{t \geq 0} \gamma^t R_{t+1}$$

where $0 < \gamma \leq 1$ is the *discount factor*. If $\gamma = 1$ the reward is *undiscounted*, which is only allowed if the MDP is episodic and the agent receives a reward of 0 in the absorbing states, otherwise the total reward could diverge. The discount factor expresses the preference towards earlier rewards over later ones.

A number of methods to compute policies in MDPs are based on the estimation of the expected cumulative discounted reward (the *return*), that is the reward expected from each state s following a policy π, as:

$$v_\pi(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t R_{t+1}\,|S_0 = s\right]$$

where R_{t+1} is the reward in the sequence $(\langle S_t, A_t, R_{t+1}\rangle,\ t \geq 0)$ generated by π. The function $v : \mathcal{S} \rightarrow \mathbb{R}$ is called the *value function*. Analogously to the value function we can define an *action value* function, that returns the expected cumulative discounted reward for executing each action from each state. It is usually represented as $q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. Its value is given by:

$$q_\pi(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t R_{t+1}\,|S_0 = s,\ A_0 = a\right]$$

The reward is accumulated by executing a in s and following π thereafter. Given an MDP, the agent aims at identifying the policy π^* such that $v_{\pi^*}(s)$ is maximum for each s. We denote such an optimal value function as v^* (and analogously q^* for the action value function). If we represent with Π_{stat} the set of all stationary policies, and with $r_{s'}$ the expected value of $R_{s'} \sim r(s, a, s', \cdot)$, the reward extracted according to r, v^* is defined as:

$$v^*(s) = \max_{\pi \in \Pi_{stat}} v_\pi(s) = \max_{a \in A} q^*(s, a),\ s \in \mathcal{S} \quad (1)$$

$$q^*(s, a) = \sum_{s' \in S} f(s, a, s')(r_{s'} + \gamma\, v^*(s')),\ s \in \mathcal{S},\ a \in \mathcal{A} \quad (2)$$

The value, and action value functions often cannot be represented exactly, storing a value for each state in tabular form, but have to be approximated. For continuous state spaces a tabular form is not even possible at all. Function

approximation also allows the agent to generalize between *similar* states (where the similarity depends on the function approximation involved). A popular approximation scheme is linear function approximation, where the state is encoded with a set of *features* and the approximated function is a linear combination of the features. With discrete actions a standard approach consists in employing a function approximator per action:

$$q_{\pi,a}(s) = \boldsymbol{\theta}^\top \boldsymbol{\phi}(s), \tag{3}$$

where $\boldsymbol{\phi}(s)$ is the feature vector, and $\boldsymbol{\theta}$ is the parameter vector to learn.

A popular function approximator, for which implementations are widely available, is Tile Coding with hashing [31]. In tile coding the state space is partitioned into *tiles*, where the union of a layer of tiles forms a *tiling*. A feature $\phi_i(s) \in \{0, 1\}$ corresponds to each tile, and therefore, only one feature can return 1 per tiling. It is desirable to have multiple tilings (possibly at different resolutions) slightly shifted from each other span the state space. Since the vast majority of the features returns 0, the linear function approximator of Equation 3 is fast to compute. It is also possible to group the tiles through hashing, reducing the total memory usage.

We will use Sarsa(λ) [31] for control, estimating the value function with True Online TD(λ) [28]. The action value function will be represented either in tabular form, where possible, or using tile coding with hashing. The exploration will be an ϵ-greedy strategy. With the ϵ-greedy strategy, the agent chooses the current optimal action according to q_π with probability $1 - \epsilon$, and a random action with probability ϵ.

2.2. Answer Set Programming

Answer Set Programming is a form of declarative programming, based on the stable model semantics of logic programming [18]. It is a propositional formalism, whose syntax is defined in terms of *atoms*, *literals*, and *rules*. An atom is an elementary proposition, such as `p` or `-q`, while a literal is an atom with or without negation, such as `p` or *not* `q`. A rule is is an expression of the form:

```
p, ..., q :- r, ..., s, not t, ..., not u
```

where "`:-`" is the (prolog-style) implication sign. The part of a rule on the left-hand side of the implication sign is called the *head*, while the part on the right-hand side is called the *body* of the rule. The head is a disjunction of literals, while the body is a conjunction of literals.

A set of rules forms an ASP theory. A model of an ASP theory is an Answer Set, that is a set of atoms compatible with the theory. Informally, each rule has to be interpreted as follows: if `r`, ..., `s` are in the answer set, and `t`, ..., `u` are not, then at least one in {`p`, ..., `q`} is in the answer set. For a formal definition of the stable model semantics we refer to Gelfond and Lifschitz [12]. The symbol *not* is often referred to in the literature as *negation as failure*. The classical negation of an atom `p` is denoted with `-p`, and is another atom, with

the implicit constraint that if `p` is in the answer set then `-p` cannot be in the answer set, and vice versa.

A *choice* rule is a particular rule whose syntax is as follows:

```
{p, ..., q} :- r, ..., s, not t, ..., not u
```

and means that if the body is verified by the answer set, then any number of atoms (including zero) in $\{\mathrm{p}, \ldots, \mathrm{q}\}$ may be in the answer set.

Two other important special cases of rules are *facts* and *constraints*. A *fact* is a rule in which the body is missing, and is denoted by omitting the implication sign:

```
p
```

means that `p` has to belong to every answer set. A *constraint*, on the other hand, is a rule in which the head is missing:

```
:- q,
```

and it means that `q` cannot belong to any answer set.

Non-monotonicity. Answer set programming is a *non-monotonic* formalism. In monotonic formalisms, such as classical logics, if a set of rules entails a particular preposition, it is also entailed by any superset of the rules. Representing entailment with the symbol $\models$, we can give the following definition of monotonicity.

Definition (Monotonic formalism)**.** *Let R be a set of logical formulas in a formalism $\mathcal{F}$, and ϕ be a logical formula. The formalism $\mathcal{F}$ is monotonic iff:*

$$R \models \phi \Rightarrow R \cup \psi \models \phi \tag{4}$$

for any formula ψ.

That is, adding formulas does not restrict the set of logical sentences that are entailed (hence, the monotonicity).

Non-monotonic reasoning allows expressing *defaults*, that is sentences that *typically* hold unless evidence to the contrary is provided. The technical element which introduces non-monotonicity into ASP is the negation as failure.

The theory of non-monotonic logics is extensive and complex. In this context, we provide only a simple example in first-order logics, which helps us illustrate later how we make use of defaults in planning. We want to represent in our model the optimistic assumption that all doors are open, and that d is a door.

$$\begin{aligned} \forall D, door(D) \quad &\Rightarrow open(D) \\ door(d). \quad & \end{aligned}$$

From these formulas, it is possible to derive that d is open. Imagine that a new piece of information becomes available at this point, namely that the door d is,

in fact, an automatic door which closes by itself, and is, therefore, closed unless held open:

$$\begin{aligned} \forall D, automatic(D) \ \wedge \ \neg held(D) \quad &\Rightarrow \neg open(D) \\ automated(d) & \\ \neg held(d). & \end{aligned}$$

It is now possible to derive both that d is open and that it is not. Indeed, the knowledge base is now *inconsistent*, and every sentence can be derived from it. This principle is known as *ex falso quodlibet*: from falsehood, anything (follows).

The initial statement about all doors being open seems now too rigid. What we would like to say is that all doors are open, unless we have any evidence to the contrary. This is expressed in ASP through negation as failure:

```
open(D) :- door(D), not -open(D)
door(d)
```

where we denote variables with uppercase letters and constants with lowercase letters. The use of variables is only for syntactic brevity since the formalism is propositional: before solving a given ASP query, the system has to ground every rule. Grounding a rule means generating all the possible rules obtained by substituting the variables with constants in all possible ways.

The first sentence can be informally interpreted as: if d is a door, and we cannot prove that d is not open, then we can derive that d is open. In this example, we see the important difference between true negation and negation as failure. Indeed, if we interpreted `not` as true negation, and substituted the double negation with the positive $open(D)$, we would obtain a tautology, which does not add anything to the knowledge base. The new information about the door being an automatic one can then be added as:

```
-open(D) :- automatic(D), -held(D)
automatic(d)
-held(d).
```

Since it *is* now possible to derive `-open(d)`, it is not possible to derive `open(d)` anymore, hence no inconsistency has been introduced.

2.3. Planning in Answer Set Programming

We want to represent a transition system $D_m = \langle \mathcal{S}_m, \mathcal{A}, f_m \rangle$ (where the subscript stands for *model*) in answer set programming, where $\mathcal{S}_m$ is a set of states, $\mathcal{A}$ is a set of actions and $f_m : \mathcal{S}_m \times \mathcal{A} \to \mathcal{S}_m$ is the transition function. This representation has to allow for fast planning, therefore in our models the set $\mathcal{S}_m$ is always discrete, and the function f_m is always deterministic.

DARLING does not depend on ASP directly, and other formalisms or planners could be used. We chose ASP for a number of reasons. First, we decided to restrict ourselves to discrete deterministic models, because they are computationally less demanding than stochastic ones, and we show in Section 4.4 that

there is no advantage in using a stochastic model regarding the policies that can be learned. Second, we require a planner to be able to compute *all* solutions with certain criteria, and Clingo can be used as one such planner while most others can only return a single plan. Third, ASP allows the designer to specify constraints on plan trajectories: an important feature as will be explained later, to limit the number of plans processed. Lastly, ASP is a *non-monotonic* formalism. Non-monotonicity lets us establish certain *defaults*, which result in very compact theories, and allow for *optimism in the face of uncertainty* which is fundamental to favor exploration.

The set of states is represented in terms of predicates whose truth value may change at different time steps and that, for this reason, are called *fluents*. A fluent is a predicate such as `at(roomA,0)` in which the last parameter is always the time step to which the predicate refers. In ASP, a fluent may appear in an answer set, appear negated in an answer set, or not appear at all. Given a finite set of fluents, $\mathcal{F}$, the set of states $\mathcal{S}_m = 3^{\mathcal{F}}$ is the set of all possible assignments (positive, negative, or unknown) to the fluents in $\mathcal{F}$. Therefore, a set of fluents uniquely determines the state space of a transition system.

Actions are represented in the same way as fluents, and are syntactically indistinguishable from them. For instance, `openDoor(d1,0)`, is an atom that means that the action `openDoor` is executed on door `d1` at time step 0.

The transition function f_m is represented in the ASP theory by determining how fluents are carried over from one time step to the next by actions, or just by the passage of time. We mentioned above that rules containing variables have to be grounded before the solver is invoked. An important variable is the time step, which is grounded up to a constant `n`, the maximum time step that will be considered.

The pre-condition of actions, that is, what must be true in a state for an action to be executable, is represented with constraints. For instance:

```
:- openDoor(D,I), not facing(D,I), door(D), I=0..n
```

means that it is not possible to execute action `openDoor(D)` on any door `D` at time step `I` and `facing(D)` cannot be derived at time step `I`. This means that `facing(D)` is a pre-condition for the action `openDoor(D)`.

The effect of actions is represented with rules such as:

```
open(D,I+1):- openDoor(D,I), door(D), I=0..n
```

which means that executing the action `openDoor(D)` at time step `I` causes the door to be open at time step `I+1`. An example of a full domain implementation is reported in the Appendix.

As introduced above, ASP allows the representation of *defaults* as soft constraints on the presence of individual atoms in an answer set. For instance, the rule:

```
open(D,I+1):- open(D,I) not -open(D,I+1), door(D), I=0..n-1
```

means: if `open(D,I)` is in the answer set, and `-open(D,I+1)` is not in the answer set, then `open(D,I+1)` is in the answer set. Through negation as failure, this rule specifies that the door `D` remains open unless some other rule imposes it to be closed, that is it remains open by *default*. In the case of a robot, it is also possible that the theory has certain properties carried over by default, but some perceptions contradict them. For example, we can specify that the position of the robot is *inertial*, that is by default it does not change:

```
at(L,I+1):- at(L,I), not -at(L,I+1), location(L), I=0..n-1
-at(L2,I) :- at(L1,I), L1 != L2, location(L1), location(L2),
I=0..n-1.
```

The second line expresses that if the robot is at a certain location, it is not anywhere else. It may happen, however, that the localization module all of a sudden realizes that the robot is at a different location, even though it was not moving. We may have:

```
at(office,0)
at(lab,1)
```

even though no action has been taken. The localization module generates the observation `at(lab,1)` and it does not cause any inconsistency, simply `at(lab,1)` causes `-at(office,1)` to be derived, which prevents the inertial law to carry over the fluent `at` by default. On the robot, with real noisy data, this ability to incorporate unexpected but acceptable pieces of information is particularly convenient. At the same time, certain observations may be deemed unacceptable and discarded. For instance, should the localization module output:

```
at(office,1)
at(lab,1)
```

this would cause the ASP theory to be unsatisfiable, therefore, this observation according to which the robot would be at two locations at the same time has to be discarded as a localization error.

Non-monotonicity, therefore, proved to be helpful to us in at least two ways: by allowing the specification of optimistic defaults, in the absence of more specific knowledge; and in being able to distinguish between constraints that should generally be assumed but may be violated, and others that must always hold.

Planning Problem. A planning problem $P = \langle D_m, s_0, \mathcal{G} \rangle$ is a tuple where D_m is a transition system, s_0 is the initial state, and $\mathcal{G}$ is a set of goal states. The initial state is specified through a set of facts about the time step 0, while the goal state is specified through constraints excluding, from the possible set of states at the last time step, all the states that do not fulfill a given goal. The ASP reasoner grounds all the rules, and generates a theory up to a given time step n, whose answer sets are all and only the histories of the form $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$, where $s_n \in \mathcal{G}$. The sequence of actions $p = \langle a_0, \ldots, a_{n-1} \rangle$ is a plan that achieves a goal state in $\mathcal{G}$.

3. Related Work

Many different types of reasoning have been referred to as planning in the literature. For the purpose of this article, we consider planning as any lookahead reasoning process that makes use of a known model of action effects to select actions. A wide variety of action representations have been considered, ranging from purely atomic representations as employed in Dyna [30] to first order relational representations as used by STRIPS and its derivatives. For the purpose of this article, we consider planning as being in contrast to learning, by which we mean either selecting actions without reference to an action model or learning the action model itself. Furthermore, planning is a purely cognitive activity, which does not involve acting in the environment, while learning is based on experience gathered by performing actions.

Automated planning and reinforcement learning both aim at solving an important class of decision-making problems. Their intersection has been considered for different purposes, using different techniques, and from various perspectives. This section gives an overview of previous work in the area of combining the model-based lookahead of planning with the hindsight view of learning.

Planning has been part of reinforcement learning since its inception, particularly in *model-based* RL, where the agent learns a model of the environment to plan subsequently on it. Dyna [30] is one of the earliest of such systems, in which a reactive component is responsible for choosing actions in real time, while the system computes a value function from both real experiences and simulated ones. Abbeel et al. [1] proposed a method close in motivation to ours, in which the agent uses a crude model of a deterministic dynamical system to identify a direction for policy improvement, and then executes trials in the real world along that direction only, updating the model with the data from the environment. All of the work on model-based RL is driven by the acknowledgment of the fact that real experience is both too expensive, to the extent necessary for achieving an optimal behavior, and potentially dangerous for physical agents. By building a model, and generalizing from experience, agents can learn from simulated actions as if they had been executed in the real world. The premise for model-based RL is shared by this work too. We also acknowledge, however, that *all models are wrong, but some are useful* [3]. Therefore, we do not rely on the model to compute the optimal policy, but only to constrain the MDP in which the agent acts. This constraint allows the agent to make use of a much simpler model, in particular ignoring probabilities and costs which are considered in model-based RL, and can be difficult to estimate. In this work we do not have the agent learn the model. Therefore, we take a perspective more characteristic of the planning community. Nonetheless, nothing prevents the model used in our method to be learned or automatically adjusted. What sets our work apart from model-based RL is how that model is used (and only partially trusted), that is, to limit the MDP of the task and not to solve it.

The way in which we constrain the search space for an optimal policy is by computing a partial policy on our model. Similar technique is used by Pinto and Fern [25] in planning, in order to reduce the dependency of sample based

planners on the branching factor. In their work, the planner learns the partial policy from previous instances of planning problems in a given domain, as a form of domain-specific knowledge to exploit in further instances. That work is an example of a line of research in combining machine learning and planning which has been fostered by the introduction of the learning track in the International Planning Competition from 2008 [10]. The emphasis is posed by the planning community on more effective domain-independent planners, while the execution of those plans in an actual environment is not taken into account. On the other hand, our focus is on the robustness of the agent's behavior, and on the adaptivity to a changing environment. Differently from the aforementioned work in planning, we use a representation simple enough to compute a partial policy from a *single* planning instance, and perform reinforcement learning on the execution of that task in the world. Learning to plan faster from previous tasks is independent of our approach.

Envelope-based planning [6] is another method introduced to limit the search space given time constraints, and it is a pure planning technique which does not involve learning. A first plan is generated through a depth-first search on a stochastic model, considering action outcomes in decreasing order of probability. An initial policy is defined over this plan, and the domain of such a policy (that is, the states over which it is defined) is called the *envelope* of the policy. A reduced MDP is then constructed over the envelope, where the actions that lead outside of it are redirected towards a special absorbing state *out*. The algorithm then proceeds in turns, in which the envelope is first expanded by adding the states reachable from the envelope in one step that are most likely to provide a policy improvement, and then computing the new optimal policy for the extended MDP. Given enough time, the algorithm extends the envelope to the full MDP and computes an optimal policy for the original problem. Our method is reminiscent of Envelope-based planning in the definition of a reduced MDP, but it differs from it on how such an MDP is computed. Instead of growing the MDP around a single plan, we define it from a set of plans, which can reach (within a particular plan length) non-contiguous areas of the MDP. We do not assume an accurate probabilistic model, and therefore do not solve this MDP through planning but RL in the actual environment. We expand the MDP at run time, and only if the agent falls outside of the set of states initially computed. State TArgeted R-MAX (STAR-MAX) [19] is a reinforcement learning approach related to envelope-based planning, in which the exploration is limited to a given envelope. In STAR-MAX, the authors do not provide a general way to compute the envelope, showing that the agent can learn one from demonstration, obtain one by simply dropping a given percentage of the states, or just rely on domain knowledge by a human to provide the envelope to the agent. The envelope only constraints the states, while our partial policy also constraints the actions (the constraints on the state space are, in our case, a consequence of some regions becoming unreachable).

Constraining the behavior of Reinforcement Learning agents is also a critical aspect of Hierarchical Reinforcement Learning [2]. In Hierarchy of Abstract Machines (HAM) [23] a manually specified state machine is employed at each

layer as a partial policy to constrain the choices available at the lower layer of the hierarchy. The lowest level of such a hierarchy is the MDP of the environment, and the application of the levels above it define a Semi-Markov Decision Process (SMDP). Leonetti et al. [16] introduced a method to generate a HAM from planning, which is a precursor of the one presented here. The only other work relating to both our own work, and constructing a hierarchy for RL from planning, is the one by Ryan [26]. Ryan used teleo-reactive operators [22], and planning with means-end analysis [20], to generate a tree with all possible operators from the current state to the goal state. At each step, more than one operator can be active. Which operator to execute is learned hierarchically, together with the underlying behavior associated with each operator. This architecture was previously introduced in RL-TOPS [27], where planning is used to generate a single plan that dictates which lower-level behavior to activate at any given time. The system introduced by Ryan [26] is similar to ours in that it computes all possible (shortest) plans to a given goal to choose which low-level behavior to activate. It is different, however, in that its aim is to construct a hierarchy and not to make the planned behavior robust. As a consequence, it does not consider the possibility of the model being incorrect, and does not compute any plan longer than the shortest plan. We show that considering plans that are longer in terms of number of actions is necessary, because they might in practice have a lower cost than the shortest plans, or be able to get the agent out of a dead-end.

Another area related to combining symbolic reasoning and RL is Relational Reinforcement Learning (RRL) [7, 32]. The aim of RRL is making use of a relational, first-order, representation to generalize through logic induction. It is particularly powerful in domains that can be naturally expressed in terms of objects and relations among them. For instance, in the well-known blocks world, the policy or the value function can be computed independently from the number of blocks. Even if generalizing through first-order lifted inference, RRL methods still calculate values and policies for the full domain, while our method uses Answer Set Programming (which is a propositional formalism) to reduce the portion of the region to explore. Generalizing through symbolic reasoning is not one of the objectives of this work, even though it is a possible interesting extension.

Lastly, planning has been combined with RL through *reward shaping* [8, 14]. Reward shaping is a technique to hasten Reinforcement Learning when the reward is sparse, and the agent has to execute a long sequence of actions before getting any feedback about its choices. The reward function is enriched by adding a term which provides feedback for intermediate states, helping guide the agent towards the goal. Grzes and Kudenko proposed a method [14] in which the agent computes a plan on a STRIPS representation of the domain, and uses it to define a shaping function. The function guides the agent along the plan, helping it find the goal sooner. Reward shaping does not constrain the behavior of the agent and, if the shaping function is potential-based, it does not alter the optimal policy [21]. As a consequence, the agent still has to explore the whole environment to learn the optimal policy, even though it can do so more

quickly. In large environments, and in particular with physical agents, it may be infeasible. In contrast, we do prevent the agent from exploring the whole domain, selecting the area to explore through automated reasoning. Our method has the potential cost of never discovering the optimal policy, but in exchange for the advantage of learning good behaviors quickly. It is particularly effective when the domain is such that reaching the optimal policy in an acceptable amount of time is impossible in any case.

4. Method definition

We assume that the domain can be modeled as an MDP $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$, where $\mathcal{S}$ can be either discrete or continuous, $\mathcal{A}$ is finite, and f and r are stochastic and unknown. The MDP D is the one for which an RL method would learn a policy to act in the real domain. Furthermore, we expect the designer to create a model of D which we will denote as $D_m = \langle \mathcal{S}_m, \mathcal{A}, f_m \rangle$. This model may or may not include the reward function. In the rest of this article, we will only use models that do not. The model D_m is the one on which a planning algorithm would compute a plan to execute in the real domain. A planning algorithm cannot plan on D since the transition function of D is unknown (and so is the reward function). We also require the designer to specify a mapping $o : \mathcal{S} \to \mathcal{S}_m$ which for every state in $\mathcal{S}$ returns a corresponding state in $\mathcal{S}_m$. In order to make sure that what is computed on the model is indeed executable in the environment, the mapping function o has to preserve the applicability of actions, that is:

$$a \in \mathcal{A}(s_m) \Rightarrow a \in \mathcal{A}(s), \forall s \in o^{-1}(s_m), s_m \in \mathcal{S}_m,$$

where by $\mathcal{A}(s)$ we denote the subset of actions of $\mathcal{A}$ applicable in state s (cf. Section 2.1). Through our method we will define a new MDP D_r (where the subscript stands for *reduced*), computed by planning over D_m, that models the same domain as D but is a reduced version of it. We will then let the agent perform RL over D_r.

In order to illustrate DARLING we will make use of a running example, a grid world shown in Figure 1. We will denote the squares starting from $\langle 0, 0 \rangle$ for the bottom left corner to $\langle 19, 19 \rangle$ for the top right corner. The agent has to navigate from the start state marked with S ($\langle 10, 0 \rangle$), to the goal state marked with G ($\langle 10, 10 \rangle$). The darker and thicker lines are walls while the red line is a door. The actions available are moving in one of the four cardinal directions and can be applied wherever they would not lead into a wall or outside of the grid. The actions are deterministic, except for the action that goes through the door, which fails if the door is closed, and the agent stays in its current state. The cost of actions depends on the state the action leads into, and will be specified with the full details of the domain in Section 5.2.1.

DARLING is composed of three steps:

1. plan generation: generate all the plans that have a cost lower than a given threshold in the model (cf. Section 4.1);

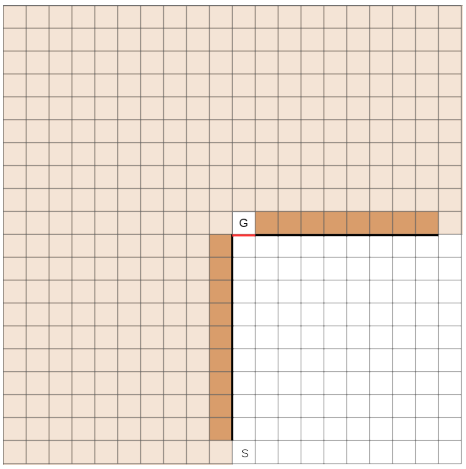


Figure 1: Example domain

2. plan filtering and merging: exclude the plans that are certainly suboptimal, and merge the remaining ones into a *partial policy* (cf. Section 4.2);
3. execution and learning: during execution, choose only actions that are returned by the partial policy, and learn their expected cumulative long-term reward in order to converge to the optimal solution (cf. Section 4.3).

4.1. Plan Generation

The plan generation step requires a planner able to return all plans that can reach the goal from the initial state within a certain cost threshold. DARLING does not depend on any specific planner or model type, and the designer should find a good compromise between the expressive power of the model and planning time. Thanks to the subsequent learning phase, the accuracy of this model is less critical than if planning were used with no learning. For this reason, in all our experiments we chose to use a symbolic, discrete, deterministic model of the environment, which does not represent action costs. Therefore, as the metric for the threshold, we used plan length.

We used Answer Set Programming to represent our models, and the answer set solver Clingo [11] to reason and plan. Planning is a particular type of reasoning possible with Clingo, where other reasoning tasks are, for instance, execution monitoring, and plan repair. We make use of non-monotonic reasoning by assuming certain system dynamics by *default*, and reconcile this assumption with perceptions during execution. For instance, the door in our example will optimistically be assumed to be open, unless the agent knows otherwise.

Optimistic assumptions are instrumental to simplifying the model, and lowering planning time: instead of considering all possible outcomes of an action,

the reasoner will take into account only the default one. Clearly, having good defaults is critical to reduce the chance of replanning. While defaults could be learned, that falls outside of the scope of this article. We will show, however, that the learning phase alleviates the problem of an incorrect default, and reduces the burden on the correctness of the model.

In ASP, plans are computed in order of length. We take advantage of this feature to define the threshold of plan length as a function of the length of the shortest plans. Since the length of a plan is not an accurate estimation of its cost, we also consider plans that are longer than the shortest plans, because they can, in fact, have a lower cost. Considering plans that are suboptimal according to the model would be necessary even if we did model costs, as there is always the possibility that the model is incorrect. Therefore, it is always advisable to generate plans that are suboptimal up to some extent, according to the model, and let the learning phase verify which one is the optimal plan in practice.

Planning

Given an initial state and a goal, we want to compute all the plans of length less than a certain threshold. We define the threshold as a function of the length of the shortest plans, but any other definition is valid. Let l be the length of the shortest plans. We want to generate all the plans of length at most $L = l \cdot \mu$ where $\mu >= 1$ is a parameter which gauges the sub-optimality the designer is willing to consider regarding the number of actions. The system first generates the shortest plans, it computes L, and then generates all the plans of length up to L.

In ASP, differently from most planners including state-of-the-art ones from the International Planning Competition, it is possible to compute all the plans of a given length. We take advantage of this feature and do not contribute reasoning or planning methods, using Clingo as a black box. With respect to our first implementation of using automated reasoning for generating finite-state controllers, based on Linear Temporal Logics [16], ASP allowed us to scale to much larger domains, and to take advantage of non-monotonic reasoning.

Given an encoding of the initial state and the goal, all the answer sets returned for the maximum time step incrementally increased up to L are all the plans of length at most L. All the queries with different maximum lengths up to L would be separate, but the answer set solver Clingo is particularly efficient at this computation. It has an incremental version of the solver designed specifically for planning, which efficiently concatenates these queries for different plan lengths retaining the fluents grounded at the previous iteration. The iterative solver makes Answer Set Planning [17] comparable in terms of planning time with state-of-the-art heuristic planning.

The answer set solver returns sequences $\langle s_i, a_i \rangle$ where s_i is a state, that is a set of fluents, and a_i is the action executed in state s_i, at time step i. Let $\mathcal{P}$ be the set of shortest plans of length l, we denote with $p = \{\langle s_i, a_i \rangle\} \in \mathcal{P}$ where $i = 0..l-1$, a plan in $\mathcal{P}$. We then augment $\mathcal{P}$ with all the plans of length up to $L = l \cdot \mu$. In our example, the only shortest plan has length 10 (ten `north`

actions), there are 0 plans of length 11, 71 plans of length 12, and so on. In every domain considered in this article, we set $\mu = 1.5$.

4.2. Plan Filtering and Merging

While every action in one of the shortest plans is necessary by definition, when forcing the length of the generated plans to be greater than the length of the shortest plans this may no longer be true. The planner can add actions that are irrelevant to reach the goal, and therefore would just sidetrack the agent. For example, in our grid world a plan might contain a cycle, or if an action `paintwall` were available, it could add such an action to the plans, even if it was not necessary to achieve the goal. A necessary sequence of actions, on the other hand, is one such that, if removed, makes the remaining plan invalid.

Having useless actions in the plans does not affect the correctness nor the asymptotic performance of the method: the learning phase will eventually avoid suboptimal actions. Nonetheless, one of the main advantages of the proposed approach is to reduce the exploration in the reinforcement learning phase significantly. Hence, we want to get rid, at planning time and as much as possible, of actions that do not contribute to reaching the goal. Limiting the exploration is critical for physical agents, where taking actions in the real world is expensive, and it is worth spending some computation on reducing the set of actions to execute.

The most effective way to eliminate unnecessary actions, particularly in ASP, is to specify constraints on plan trajectories as part of the goal, and avoid useless actions to be generated in the first place. For instance, in our example we could add the rule:

```
:- pos(X,Y,I1), pos(X,Y,I2), I2 > I1.
```

which means that it is not possible for the agent to be at the same location at two subsequent time steps.

This solution shifts the burden to the user, however, and we want the system to perform well even when the user cannot, or does not want to, express such constraints as part of the goal. Therefore, we introduce an algorithm for plan filtering which discards plans for being certainly suboptimal.

We begin by defining which plans are acceptable, and which ones must be discarded. For the following definition, we consider plans as sequences of actions, and ignore the intermediate states.

Definition (Redundant plan)**.** *A plan $p = a_1 \dots a_l$ is* redundant*, if there exists a sequence of actions $p_r = a_m \dots a_n$ with $1 \leq m \leq n \leq l$ such that p can be divided into three concatenated sequences $p = a_1 \dots a_{m-1} p_r a_{n+1} \dots a_l$ and $p' = a_1 \dots a_{m-1} a_{n+1} \dots a_l$ is a valid plan.*

Definition (Minimal plan)**.** *A plan that is not redundant is said to be* minimal*.*

Ideally, we would like to reject every redundant plan. For instance, it is easy to see that plans that contain a cycle are redundant. Since Clingo generates

plans in order of increasing length, once they are stored in memory checking if a sequence of actions is a plan does not require any new invocation of the reasoner. Reasoning, and in particular planning, is the most computationally expensive process in our approach. Hence, as long as our filtering of redundant plans is not the computation bottleneck, the scalability of this method is only limited by the planner.

As an example of a filtering problem, consider the plan `aaabdca` where each letter is an action. If a filtering algorithm is processing a plan of length 7, it has already either accepted or rejected all plans of length up to 6. Let `aaaa`, and `aabaca` be two accepted (minimal) plans, and let `aabcaa` be a discarded plan. In order to check whether `aaabdca` is redundant, by applying the definition, an exhaustive algorithm would have to remove every subsequence of actions systematically from the plan, and test whether what remains is also a plan. Since the shortest plan has length 4, the algorithm can stop after having tested all subsequences that result from removing at most 3 actions. In this example, there exists the sequence `bdc` such that what remains after its deletion, `aaaa`, is a plan, therefore `aaabdca` is redundant. In order to achieve this conclusion, the exhaustive algorithm has to test 17 subsequences: 7 obtained by removing one action, 6 by removing 2 actions, and terminate on the 4-th by removing three actions.

We store the plans in lexicographical order, so that for P plans of length between l and L verifying whether a sequence is a plan is $\mathcal{O}(L \log P)$. We only need to remove at most $L - l$ actions, since otherwise the resulting sequence would have fewer actions than the shortest plans. Checking all subsequences of every plan requires $\mathcal{O}((L-l)^2)$ search operations per plan, with a worst-case asymptotic complexity of $\mathcal{O}(P(L-l)^2 L \log P)$. It is clear from this expression that if $L = l$ no filtering takes place, as expected, since all plans are shortest plans.

While tractable, the exhaustive algorithm has proved to be too expensive in our robotic domain. In order to make DARLING more practical, we devised an algorithm for plan filtering which has the same worst-case asymptotic performance as the exhaustive search, but is about twice as fast on average. The algorithm (shown in Algorithm 1) performs two tests on the plans, exploiting insights on different types of possible redundant plans.

All plans are stored in a sorted set, ordered lexicographically. The plans are considered in blocks of increasing length, starting from $l + 1$. The first test performed (line 7) is on whether they contain cycles. Containing cycles is the most common reason for which plans are discarded, therefore this check of complexity $\mathcal{O}(L \log L)$ discards most of the redundant plans.

The second test is based on the presence of a *suspicious* action, that is, the first action that does not belong to any minimal plan. The algorithm that returns the suspicious action is shown in Algorithm 2. Such an action is identified with a look-up in the sorted set of minimal plans (line 1) for the position where the current plan would be inserted. Either the plan immediately before or immediately after that position is the plan with the longest prefix of actions in common with the current plan. Both prefixes are computed (line 3 and 4),

Algorithm 1: FastPlanFiltering

```
   Data: all_plans, list of plans, sorted by plan length
   Result: minimal_plans, list of accepted plans
 1 redundant_plans ← ∅;
   // list of pointers to discarded plans in all_plans
 2 l ← length of the shortest plan;
 3 L ← length of the longest plan;
 4 for i ← l + 1 to L do
 5     current_plans ← plans in all_plans of length i;
 6     for p ∈ current_plans do
           // if the plan contains a loop it is immediately
              discarded
 7         if hasLoop(p) then
 8             redundant_plans ← redundant_plans ∪ p;
 9         else
               // the first action that does not belong to any
                  minimal plan
10             a ← firstSuspiciousAction(p, minimal_plans);
11             bad ← false;
12             j ← 1;
13             while j ≤ i − l ∧ ¬bad do
14                 bad ←
                   checkSectionWithLength(p,a,j,minimal_plans,redundant_plans);
15             if bad then
16                 redundant_plans ← redundant_plans ∪ p;
17     for p ∈ current_plans ∧ p ∉ redundant_plans do
18         minimal_plans ← minimal_plans ∪ p;
19 return minimal_plans
```

Algorithm 2: FirstSuspiciousAction

```
   Data: p, plan currently analyzed
   minimal_plans, list of accepted plans
   Result: suspect, suspicious action
1  lb ← lowerBound(p,minimal_plans) ;
   // pointer to first element in minimal_plans which is not
      considered to go before p
2  lbb ← plan before lb in minimal_plans;
3  l1 ← matchingPrefix(p,lb) ;
   // length of the matching prefix
4  l2 ← matchingPrefix(p,lbb) ;
5  if l1 > l2 then
6  |  chosen = l1 + 1 ;
7  else
8  |  chosen = l2 + 1 ;
9  return pointer to action in p at position chosen
```

and the first action that does not belong to the longest prefix is returned as the suspicious action.

We clarify this algorithm with an example. Let `aaabdca` be the plan the algorithm is testing, and let the insertion point of this plan in the ordered list of minimal plans be as follows:

```
 ...
aaaa
← aaabdca
aabaca
 ...
```

The lower bound of `aaabdca` in this set is `aabaca`. Between the two plans immediately preceding and following the insertion point, the one that shares the longest prefix with the plan we are testing is `aaaa`. The matching prefix has length 3, and the action at position 4, that is `b`, is the suspicious action of this plan.

At line 14, Algorithm 1 tests if any subplans of length at least l that do not contain the suspicious action are valid plans. This algorithm is shown in Algorithm 3. CheckSectionWithLength is invoked to remove sequences of actions of increasing length, starting from 1 up to $i - l$ where i is the size of the current plan, and that contain the suspicious action. It creates a sliding window from `initial_pos` (line 2) to `final_pos` (line 5) of `length` actions around the suspicious action, and tests the plan without that window (lines 8 and 10). In our example, it removes the sequences: `b`, `ab`, `bd`, `aab`, `abd`, and `bdc`. At `bdc` it stops, since the remaining plan is `aaaa` which is a valid plan (it is in the list of minimal plans), therefore our test plan is redundant and must be discarded.

Algorithm 3: CheckSectionWithLength

```
   Data: current_plan, the plan being tested
   suspicious, the index of the suspicious action
   length, the length of the string of actions to remove
   good_plans, the minimal plans identified so far
   bad_plans, the redundant plans identified so far
   Result: true if current_plan is certainly redundant
 1 dif_pos ← max(−(length − 1), −suspicious);
 2 initial_pos ← suspicious + diff_pos;
   ;                              // does not go past the first action
 3 s ← length of current_plan;
 4 while initial_pos ≤ suspicious ∧ initial_pos + length ≤ s do
 5 |   final_pos ← initial_pos + length;
 6 |   test_plan ← current_plan [0, initial_pos);
   |   // plan concatenation
 7 |   test_plan ← test_plan + current_plan [final_pos, s);
 8 |   if good_plans contains test_plan then
 9 |   |   return true;
10 |   if bad_plans contains test_plan then
11 |   |   return true;
12 |   initial_pos ← initial_pos + 1;
13 return false;
```

Theorem 1. *Algorithm 1 is correct, in that it discards only* redundant *plans.*

Proof. Algorithm 1 discards plans under two conditions: (1) they contain a loop, and (2) it could identify a sequence of actions such that what remains after removing it is a plan. The second condition is verified by checking that the subplan constructed at line 7 of Algorithm 3 belongs to either the set of good plans (line 8) or bad plans (line 10). This condition is correct by definition of redundant plan. If a plan contains a loop, it is of the type: $\langle s_0, a_0\rangle$, ..., $\langle s_i, a_i\rangle$, ..., $\langle s_j, a_j\rangle$, ..., $\langle s_{l-1}, a_{l-1}\rangle$, where $s_i = s_j$. It is possible to identify the sequence of actions $a_i, \ldots, a_{j-1}$ such that if removed the remaining plan $\langle s_0, a_0\rangle$, ..., $\langle s_j, a_j\rangle$, ..., $\langle s_{l-1}, a_{l-1}\rangle$ is a valid plan, which makes the original plan redundant. □

We cannot prove that Algorithm 1 is also complete, that is, that it discards *all* redundant plans. Its performance in terms of both redundant plans retained and computation time will be experimentally compared with the exhaustive algorithm in Section 5.

Plan Merging

All minimal plans that pass the previous filtering tests are merged into a *partial policy*. Recall that a policy in an MDP is a function that returns the action to execute for each state (cf. Section 2.1). A partial policy, in this context, is a function $\pi : \mathcal{S} \to 2^{\mathcal{A}}$ that maps a state into a *set* of possible actions. We define this function formally as follows: let $\Pi(L)$ be the set of minimal plans of cost (in our case length) up to L for a given planning problem, then

$$\pi_L(s) = \{a | \exists p \in \Pi(L) \text{ s.t. } \langle o(s), a\rangle \in p\},\ \forall s \in \mathcal{S} \tag{5}$$

is the partial policy that returns, for each state in the MDP, the set of actions that belong to at least one minimal plan in the corresponding state of the model. The function $o : \mathcal{S} \to \mathcal{S}_m$ is the mapping from the states of the MDP to the states of the model introduced at the beginning of this section. This partial policy is used in the reinforcement learning phase to define an MDP on which the agent learns the optimal policy. The partial policy is defined over the state space of the original MDP, to allow the learning algorithm to adapt to the real environment, and not to the model.

We visualize the partial policy computed with $\mu = 1.5$ in Figure 2. Out of the 400 cells part of the original MDP, only 53 are reachable in the reduced one. The number of executable actions in the original domain is 1526, while only 106 are available to the agent in the reduced domain.

4.3. Execution and Learning

The partial policy computed in the previous phase is used during the execution to constrain the agent's behavior. At each state in the original MDP D, the agent chooses an action from among the ones returned by the partial policy for that state. Making an informed choice at this stage is the last step of DARLING, and is based on reinforcement learning.

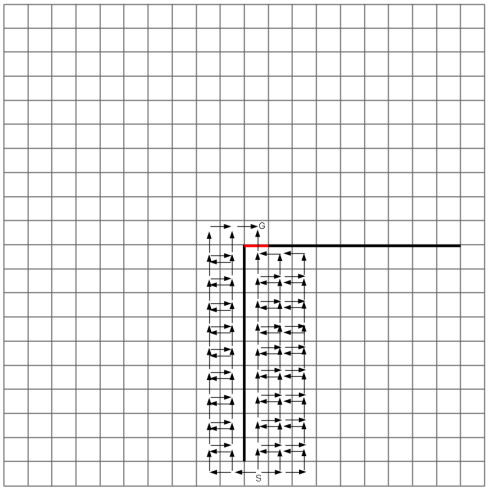


Figure 2: The actions returned by the partial policy

The partial policy may be used to define an MDP over a reduced transition function, which allows only a strict subset of the transitions. In particular, the new transition function does not permit the agent to choose any action that does not belong to at least one minimal plan. We define this reduced MDP $D_r = \langle \mathcal{S}, \mathcal{A}_r, f_r, r \rangle$, based on the original $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$ where the actions available are restricted to those returned by the partial policy:

$$\mathcal{A}_r(s) = \pi_L(s).$$

The transition function is defined from the one of D:

$$f_r(s, a, s') = f(s, a, s'),\ \forall s, s' \in \mathcal{S}, a \in \mathcal{A}_r(s)$$

but it is undefined for actions $a \notin \mathcal{A}_r(s)$.

While the partial policy (and therefore the model D_m) determines the possibility to choose a given action in a particular state, the probability distribution over the next state is unmodified for the actions that are available. Thus, an action can make the current state transition into a state that is not what D_m predicts: if the agent takes action a in state s, it can land in a state s' such that $s' \notin o^{-1}(f_m(o(s), a))$. In that case, there will be no action available in s', and the system replans, computing a new partial policy from s'. Let the old partial policy be π'_L, and the new partial policy be π''_L. The agent merges the two policies computing

$$\pi_L(s) = \{a | a \in \pi'_L(s) \vee a \in \pi''_L(s)\},\ \forall s \in \mathcal{S}$$

consequently augmenting the MDP D_r with the transitions that belong to the new plans.

The agent can learn a policy for the reduced MDP D_r, expanding it when necessary if the system transitions into a state where no action is available. The learning layer is completely independent of the generation of the partial policy, and works on the MDP it induces. Note that the learning algorithm does not learn a policy for D_m, our model. The transitions and the state space are provided by D, the real environment. Even if f and r are unknown, they can be sampled during the execution, as is common in RL. Learning through sampling in D_r provides the capability to overcome inaccuracies of the model, which allows this method to be more robust than planning alone.

Since D_r is a smaller MDP than D, it is possible that the optimal policy in D requires actions that have been removed from D_r. In that case, the behavior learned by the agent will be suboptimal in the full domain, while it is guaranteed to be optimal in D_r. In our real-world experiments, we cannot know whether this was the case since we could not find a solution for the full domain. An example of a domain in which the solution in D_r was suboptimal is given by Leonetti et al. [16] in a simulated environment. In practical cases where finding an optimal policy for D is infeasible, DARLING provides a way to compute a policy still as the result of an optimization process, over a specifically defined MDP. In practice, defining an optimization problem is often more important to have a direction for improvement, rather than to actually obtain the optimal solution.

It is possible to learn a policy in D_r with both model-free and model-based RL algorithms. In model-based RL the agent learns the model, and it may achieve a more effective generalization than a function approximator on the value function alone, which is used in model-free RL. Model-based RL also introduces an additional learning bias, on the model itself, besides the value function. Inaccuracies of models are what originally motivated this work. For this reason, we only used a model-free algorithm in our evaluation, but a model-based one could also be applied both to learning with no previous knowledge, and to learning a policy for D_r.

4.4. Robustness to Inaccurate Models

In this section, we analyze how the model D_m affects the ability of the agent to learn an optimal policy. As we mentioned at the beginning of Section 4.1, DARLING does not depend on any particular representation formalism, planner, or metric. What it does require is that the chosen formalism and planner allow the agent to compute all plans of cost at most t from the optimal policy, where t is a threshold function based on a metric that the planner can minimize. In our implementation, we used Answer Set Programming as the representation formalism, Clingo as the planner, and plan length as the metric.

We first generalize the definition of a partial policy in Equation 5 for any model $D_m = \langle \mathcal{S}_m, \mathcal{A}, f_m, r_m \rangle$ of an MDP $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$ as follows:

$$\pi_t(s) = \{a \mid a \in \mathcal{A},\ q^*_m(s,a) \geq v^*_m(s) - t(s)\},\ \forall s \in \mathcal{S}_m.$$

For the set $\pi_t(s)$ to be non-empty, $t(s) \geq 0$ must hold for every $s \in \mathcal{S}_m$. In order to scale with the reward, the function t should depend on q^*_m (or equivalently on v^*_m). For instance, in our implementation where the agent accepts every plan μ times the length of the optimal plan, t can be written as

$$t(s) = (1 - \mu)\ v^*_m(s),\ s \in \mathcal{S}_m,$$

which is positive for $\mu \geq 1$ since $v^*_m(s) \leq 0$ (it is the opposite of plan a length) for all $s \in \mathcal{S}_m, a \in \mathcal{A}$. It follows that:

$$q^*_m(o(s), a) \geq v^*_m(o(s)) - t(o(s)) \Leftrightarrow a \in \mathcal{A}_r(s),\ \forall s \in \mathcal{S},\ a \in \mathcal{A}. \tag{6}$$

That is, an action a is available to the agent in a state s if and only if its value in the model for the abstract state $o(s)$ is close to the optimal value by more than the threshold. Thus, the question of whether or not the agent will learn to execute a particular optimal action can be rephrased entirely in terms of the value of that action in the model, with respect to the threshold function.

The fact that the inclusion of an action a in the set $\mathcal{A}_r(s)$, for a given state s, depends solely on the value $q^*_m(o(s), a)$, and the threshold function t, has two consequences. First, the actual cost of actions does not play any role. Therefore, the model can be arbitrarily inaccurate in how it estimates action costs. Indeed, what the planner uses as a metric may be completely unrelated to the cost in the actual environment. Second, the transition function of the model f_m only enters the equation through the value q^*_m (cf. Equation 2). Thus, the modeled action transition probabilities do not need to be accurate, as long as they guarantee the value to be above the threshold.

It is interesting to verify that, should a stochastic model D_m be available, it is always possible to make a deterministic model in which the same optimal policies can be learned. Therefore, there is no need, if there is a computational gain in planning over a deterministic model, to use a stochastic one.

Theorem 2. *Given a stochastic model D_m it is possible to obtain a deterministic model $\hat{D}_m$ for which there exists a threshold function $\hat{t}$ such that*

$$q^*_m(s, a) \geq v^*_m(s) - t(s) \Rightarrow \hat{q}^*_m(s, a) \geq \hat{v}^*_m(s) - \hat{t}(s),\ s \in \mathcal{S}_m,\ a \in \mathcal{A}.$$

That is, the determinization does not cause the agent to lose the ability to choose that action. Not all determinizations have this guarantee, for instance selecting the most likely outcome for each action does not. For a determinization to have this property, it must be *optimistic*, which we define as follows.

Definition (Optimistic model)**.** *Given a model $D_m = \langle \mathcal{S}_m, \mathcal{A}, f_m, r_m \rangle$, a model $\hat{D}_m = \langle \mathcal{S}_m, \mathcal{A}, \hat{f}_m, r_m \rangle$ is* optimistic *with respect to D_m if and only if, given a policy π:*

$$\hat{q}_{m,\pi}(s, a) \geq q_{m,\pi}(s, a),\ \forall s \in \mathcal{S}_m,\ a \in \mathcal{A}.$$

That is, an optimistic model is one that overestimates the value of actions.

We can now prove that Theorem 2 holds if $\hat{D}_m$ is an optimistic model with respect to D_m, and show how to construct one.

Proof. Let $g_m(s,a,s') = r_{s'} + \gamma v^*_m(s')$, where $r_{s'}$ is the expected value of $R_{s'} \sim r(s,a,s',\cdot)$, be the component of the value of a state-action pair due to state s'. Equation 2 can be rewritten as:

$$q^*_m(s,a) = \sum_{s' \in \mathcal{S}'} f_m(s,a,s') g_m(s,a,s'),\ s \in \mathcal{S},\ a \in \mathcal{A}.$$

Given a state $s \in \mathcal{S}_m$ and an action $a \in \mathcal{A}$, let $\mathcal{S}'_m = \{\hat{s} \in \mathcal{S}_m \mid f_m(s,a,\hat{s}) > 0\}$ be the set of states in which the system may transition after the execution of a in s. Given a stochastic transition function f_m, an optimistic determinization $\hat{f}_m$ can be computed by choosing an outcome $s'_{s,a} \in \mathcal{S}'_m$ for each $s \in \mathcal{S}_m$ and $a \in \mathcal{A}$ which provides a return higher than what it would be under f_m, that is which satisfies:

$$g_m(s,a,s'_{s,a}) \geq q^*_m(s,a),\ s \in \mathcal{S}_m,\ a \in \mathcal{A}.$$

Such a state $s'_{s,a}$ always exists: since the function q is an average of the functions g weighted by the transition probability, at least one of them has to be greater than or equal to their average. The most optimistic model is the one in which

$$s^*_{s,a} = argmax_{s'}\ g_m(s,a,s')\ \forall s \in \mathcal{S}_m,\ a \in \mathcal{A}$$

is chosen. The deterministic model can then be constructed by setting $\hat{f}_m(s,a,s'_{s,a}) = 1$ for the chosen $s'_{s,a}$ for each state-action pair. For such a model,

$$\hat{q}^*_m(s,a) = g(s,a,s'_{s,a}) \geq q^*_m(s,a),\ s \in \mathcal{S}_m,\ a \in \mathcal{A}. \tag{7}$$

Let $\hat{t}(s) = \hat{v}^*_m(s) - v^*_m(s) + t(s)$ be the threshold function for $\hat{D}_m$. It is positive definite since $\hat{v}^*_m(s) \geq v^*_m(s)$ (which follows directly from Equation 7).

It follows that:

$$\begin{aligned}
\hat{q}^*_m(s,a) &\geq q^*_m(s,a) && (8)\\
&\geq v^*_m(s) - t(s) && (9)\\
&= t(s) - \hat{t}(s) + \hat{v}^*_m(s) - t(s) && (10)\\
&= \hat{v}^*_m(s) - \hat{t}(s)\ s \in \mathcal{S}_m,\ a \in \mathcal{A}. && (11)
\end{aligned}$$

where, line 8 is Equation 7, line 9 is true by hypothesis, and line 10 has been obtained by substituting $v^*(s)$ using the definition of $\hat{t}(s)$. □

Therefore, no action that would be selected under D_m will be discarded under $\hat{D}_m$. If a deterministic planner is preferred over a probabilistic one due to time constraints, as in our experiments, an optimistic determinization provides the highest chances of being able to learn the optimal policy. Those chances, in that case, reside solely in the choice of t, the only parameter of the method. Optimism in the face of uncertainty is a principle which has affected exploration methods for a long time [4], and it is not surprising that it can be applied to our framework as well.

Compared to Ryan's architecture [26] (cf. Section 3), the main difference is in the relaxation of the constraint for which actions are included. In Ryan's work, resilience to inaccuracies of the model is not taken into account, and the planner is used to generate optimal plans only. Therefore, for an action to be considered, Equation 6 has to be satisfied with $t(s) = 0$ for each state, which also means it is satisfied with equality. This stricter constraint imposes on the model the additional requirement that the optimal actions in the environment have to be also the optimal actions in the model, or they will be discarded. That is, the model has to preserve at least the partial ordering between the optimal actions and the other actions in every state. This does not have to be the case for DARLING, where the optimal action can have a value lower than other actions, as long as the difference is less than t. While the relaxation of the equality constraint is a simple technical difference, the implications on the usability of the method are considerable. Having to create a model in which the optimality of actions is maintained leaves little leeway for inaccuracies. For instance, in our grid-world domain, it is not possible to use plan length as a metric and have the shortest plan be optimal. A more complex model which includes action costs would be necessary, along with a more computational expensive metric planner.

Another advantage of the introduction of t, and the consequent inequality, in Equation 6 is that it makes it easier to create a model that allows the agent to track a non-stationary environment. If we consider not only the optimal actions at the moment, but also every action that can be optimal over time, the agent can track the current optimal action among the ones available, as demonstrated in the experiments of Section 5.2.1 and 5.2.2. Doing so by selecting optimal actions would require the model to have all the actions that *can* in practice be optimal, be optimal in the model at all times. Such a model would be much more difficult to design, while, given a model, this property can be achieved by choosing an appropriate t.

Clearly the choice of t is critical. Without knowing the value function of D, or how the environment can change over time, it is not possible to guarantee that an optimal policy can be learned. It is guaranteed, however, that the optimal policy of D_r can be learned. In practice, the choice of t can be dictated by a trade-off between planning time and sample complexity for the RL algorithm.

5. Experimental validation

We validate our method experimentally, first discussing the performance of the plan filtering algorithm, and then of the whole method in different domains. A software library for planning in ASP, monitoring execution and learning, which also implements DARLING, is available online[1].

[1] https://github.com/mleonetti/actasp

5.1. Plan Filtering

In Section 4.2 we introduced an algorithm for plan filtering which has the same worst-case asymptotic complexity as exhaustive search, and we claimed that it performs better in the average case. We substantiate this claim with an experiment on a grid world of the same type as the one introduced in Section 4, but of size 50 by 50.

The initial state is the same as Figure 1, with the agent starting at $\langle 10, 0 \rangle$. We added an action `changeColor(C)`, which changes the color of the cell in which the agent is to `C`. We also added four colors, doubling the branching factor of the original grid. These actions do not contribute to reaching the goal, and therefore provide a way of constructing redundant plans other than by introducing cycles.

We ran 1000 trials in which the agent picked a goal location randomly, computed the shortest plans of length l, and generated as many plans as possible of length up to $1.5l$, within a 5-minute deadline. This domain allows for plans of a length that challenge state-of-the-art planning algorithms, and computing all suboptimal plans is infeasible for some instances. Therefore, we limited the plan generation time and compared both algorithms on the set of plans computed within that limit. The agent filtered those plans 10 times, calculating the average execution time. Figure 3 shows the results of plan filtering plotted over the size of the input plan set. The length of the shortest plans varied from

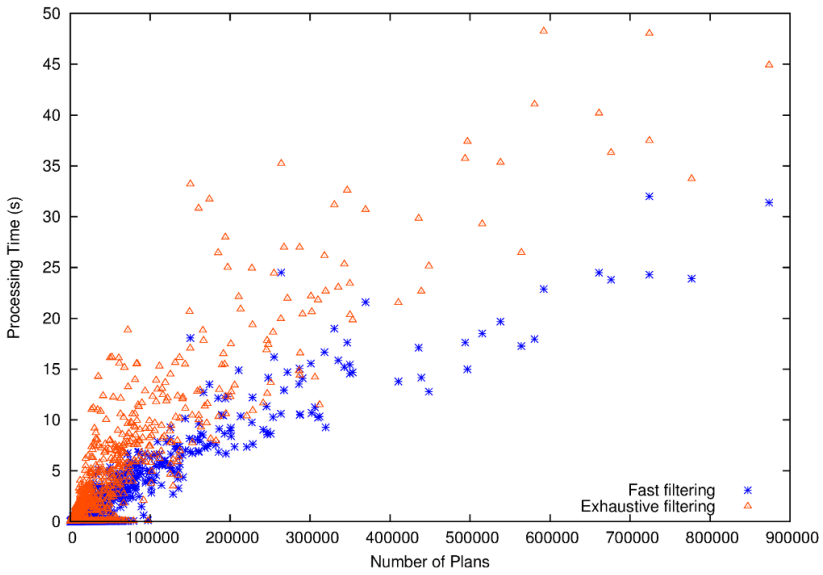


Figure 3: Comparison of computation time for plan filtering, with Fast Filtering versus exhaustive search.

1 to 44 actions. The average computation time for our algorithm (denoted as

Fast Filtering) was of 2.21s, while it was of 3.93s for the exhaustive search. Since the exhaustive search is also a complete algorithm (while we only proved correctness for ours) the set of plans accepted by our algorithm is in general a superset of the plans accepted by exhaustive search. In all instances the number of accepted plans was exactly the same, however, showing that our algorithm did not miss any minimal plan. The average speed-up of almost a factor of 2 proved indispensable in increasing the reactivity of the system, especially in the experiments with the robot. The difference between a response time of 10 seconds and one of 20 seconds often determines whether a person would walk away from the robot or not.

5.2. Learning and Execution

We carried out three experiments, in order to demonstrate different characteristics of DARLING. We compare three agents across all experiments, for each domain D and model D_m: an agent that makes decisions by planning on D_m only, an agent that makes decisions by reinforcement learning on D, and DARLING that uses D_m to compute a partial policy, and limits its exploration to D_r. We refer to the first agent as the P agent, to the second as the RL agent, and to the third as the PRL agent.

The first domain is the grid world introduced in Section 4, where we show how our agent limits the exploration with respect to the RL agent that cannot benefit from the knowledge in the initial model, and this also allows it to track the environment as it changes, thanks to a more directed exploration.

The second experiment was carried out in a 3D simulated environment, where a mobile robot navigates between different locations, under changing environmental conditions. Here we show how DARLING adapts to a more realistic environment, which proves too challenging for the RL agent. The last experiment was performed on a real mobile autonomous robot, executing different service tasks in our department for two weeks. We show how the PRL agent could complete tasks where the P agent fails and improves its performance over time.

For all our experiments we used Sarsa(λ) [31] as the learning algorithm for both the RL and the PRL agents, exploring with an ϵ-greedy strategy [31], and estimating the value function with the recently introduced True Online TD(λ) [28]. We always used $\lambda = 0.9$ and $\gamma = 1$, since all tasks are episodic. The P and PRL agents do planning with answer set programming over the same model, and for the PRL agent we always set $\mu = 1.5$ as the parameter for the threshold on plan length. The P agent always executes one of the shortest plans, chosen at random. The P agent can take advantage of more than one plan, since the ASP reasoner can return all the shortest plans. Choosing one plan randomly grants the agent a certain robustness that is not achievable with planning algorithms that can only return a single plan. Nonetheless, the P agent does not adapt over time.

5.2.1. Grid World

The first experiment was performed on the grid world of Figure 1. The state space is $\mathcal{S} = \{\langle x, y, d\rangle \mid x, y \in [0, 19] \wedge d \in \{-1, 0, 1\}\}$, where x and y are the coordinates of the agent in the grid, and d is the state of the knowledge (called the *belief*) of the agent about the door. When $d = -1$ the door is known to be closed, if $d = 1$ it is known to be open, and if $d = 0$ it is unknown.

The actions are deterministic and always succeed, except for the action that goes through the door, which fails with the probability of the door being closed, defined for episode e as:

$$p_d(e) = \begin{cases} 1 - \frac{e}{E-1} & \text{if } 0 \leq e < E \\ 0 & \text{otherwise} \end{cases}$$

where e is the episode number and E is the number of episodes during which the environment is non-stationary.

The reward is also deterministic, therefore we can denote $r(s, a, s', g) = 1$ as $r(s, a, s') = g$. For this experiment it is defined as follows:

$$r(s, a, s') = \begin{cases} -4, & \text{if } s' \in \mathcal{S}_1 \\ -2.65, & \text{if } s' \in \mathcal{S}_2 \\ -1, & \text{if } s' \in \mathcal{S}_3. \end{cases}$$

where $\mathcal{S}_1$ is the set of states in Figure 1 corresponding to the squares in dark orange, $\mathcal{S}_2$ corresponds to the region in light orange and $\mathcal{S}_3$ to the region in white. The reward was designed with two goals in mind: demonstrating resilience to details ignored in the model (action costs), and adaptivity to a continuously changing environment. The paths along the wall have a lower reward, so that the shortest path in the model is not the optimal one in practice. Furthermore, the reward is such that the value of going north in the initial state equals the value of going west at episode $e = \frac{1}{3}E$. After that point, the optimal policy switches to moving north, and we can verify how the agents adapt.

The system is always initialized in a state where the condition of the door is unknown. The agents can know the true state of the door only when at the location next to it. While this problem can be modeled as a Partially Observable MDP (POMDP) [15], here we consider as the state space the belief state of such a POMDP, which is an MDP. If unknown, the ASP model of the P and PRL agents optimistically assumes the door to be open, otherwise these agents would not even plan to try to go through it.

Knowledge Representation. The state space is represented differently at the planning and at the learning level. The ASP model for planning has two fluents `pos` and `obst` to represent the position of the agent and the presence of obstacles respectively (cf. the Appendix). The knowledge representation for learning is a tile-coding function approximator composed of two groups of tilings (cf. Section 2.1), at resolutions 5 and 2 respectively on the x and y variables and resolution 1 for both groups on the d variable. The two groups have 16 an 4

tilings respectively. This allows the agent to generalize at different levels, over positions that are 5 and 2 cells apart, with a precision of $5/16 \approx 0.31$ for the first tiling and $2/4 = 0.5$ for the second one. The number of tilings have been chosen to be powers of two, and guarantee a precision of at least 0.5. The total number of possible states of the agent's knowledge is $20 \cdot 20 \cdot 3 = 1200$. Therefore, the memory requirement of the function approximator (the total number of tiles) should be not more than that. We reduced the actual memory requirement to 512 tiles through hashing, which proved adequate to learn the task. The total memory used by the approximator in this experiment is, consequently, about half what would be necessary in tabular form.

Agents. In addition to the P, PRL, and RL agents already described, in this experiment we also introduce a number of agents which we denote as Pmem-n, where $n \in \{1, 10, 50, 200, 500\}$ is the memory length of the corresponding agent. The agent Pmem-n remembers the past n door observations, and estimates the probability of the door being closed given this data. At the beginning of each episode, it initializes the ASP model with the most likely door state. Until the n observations have been collected the door defaults to open, as with the other planning agents. This agent is a very simple instantiation of an agent which estimates transition probabilities, and adapts the ASP model towards the most likely outcome. The grid-world experiment was designed to have a single probabilistic transition, therefore the probability of the door being closed is the only parameter such an agent has to estimate.

Parameter Selection. We ran 500 trials of 800 episodes each, with a non-stationary phase of $E = 600$ episodes. The parameter $\alpha_t = 0.2$ is the learning step size for an agent that learns the value function exactly, and was selected to be as high as possible while achieving a stable behavior. To obtain the step-size parameters for the function approximator we normalized it with respect to the number of tilings $\alpha = \alpha_t/T$, with $T = 20$. We set $\epsilon = 0.2$ for the exploration strategy for both the PRL and RL agents, in addition to the parameters specified for every domain. The value of ϵ has been selected to be as low as possible, while allowing the agent to track the environment. These values have been chosen with a quick manual tuning, and are by no means optimal. Nonetheless, they are shared by all the reinforcement learning agents, allowing for a fair comparison.

Results. The performance of all agents is shown in Figure 4, with a sliding window of 5 episodes. The figure also shows the 95% confidence intervals, plotted every 5 episodes to avoid cluttering the image.

The reward of the P agent follows the probability of the door exactly, forming a linear interpolation between the minimum and maximum reward. The plan it executes, that is to go straight north and replan if the door is closed, becomes optimal after episode $E/3 = 200$. The Pmem-1 agent finds the door closed on the first episode, and it updates the model accordingly, planning to go west around the wall for the rest of the trial. This behavior may seem limited by the short memory, but we show in Figure 5 that no amount of memory can improve

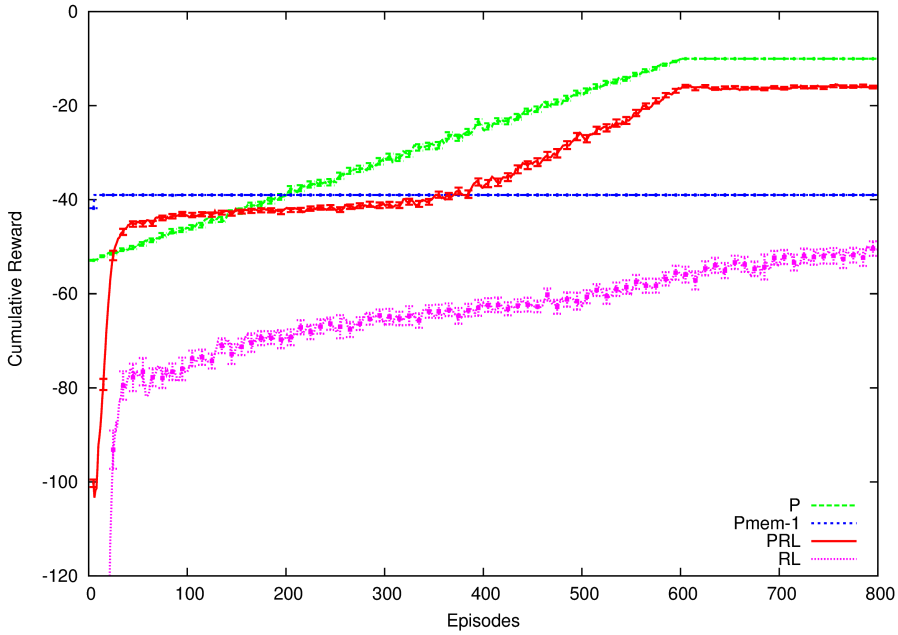


Figure 4: Comparison of the cumulative reward per episode of the agents while learning.

the behavior of the Pmem-n agents in this experiment while the environment is non-stationary. In general, such an agent is subject to blocking some path in the model and never being able to go back to it again, even if it becomes optimal. It is interesting to note that even if the Pmem-500 agent had experienced a better policy, when its memory is full it drops to a worse one. The reason for this behavior is that the decisions of the Pmem-n agents is based on the probability of the transitions and not the *value* of the actions, differently from the reinforcement learning agents.

Since the Pmem-n agents do not, in general, carry any guaranteed benefit over P agents, but require to estimate the transition probabilities of actions, we discarded them for the experiments on the more complex and realistic environments.

The PRL agent quickly learns the initial optimal policy, and then tracks the environment switching to a new policy at around episode 400. The RL agent, on the other hand, learns the initial optimal policy but is unable to discover the new one in the time allocated for this experiment. We also granted 200 episodes past the non-stationery phase to see if that allowed the RL agent to catch up, but it is still not enough. Its performance increases continuously, but too slowly with respect to the rate of change of the environment. Furthermore, the reward accumulated in the first 20 episodes is enormously lower than the other agents', and in particular of the PRL agent. The PRL agent achieves an average reward

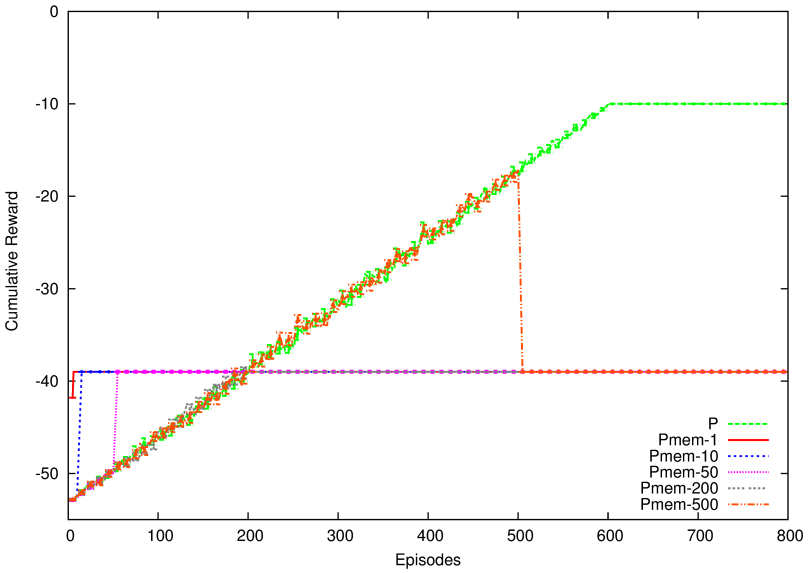


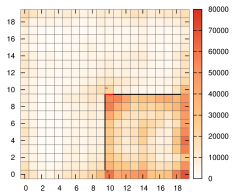
Figure 5: Comparison of the Pmem agent family.

over the first 5 episodes of about -100, while the RL agent of about -2250. The RL agent has to explore the whole grid in order to find the optimal path to the goal, and this behavior accounts for the initial low performance. We visualize the difference in the exploration of the RL and PRL agents in Figure 6.

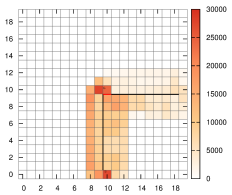
The RL agent covered the whole grid, while the PRL agent only explored a portion of the grid shown in Figure 6b.

Figure 6c and 6d show the final behavior of the agents, in which the RL agent is still executing the initial optimal policy. This initial policy returns the action `south` for all the states between $\langle 10, 1\rangle$ and $\langle 10, 8\rangle$ in which the optimal action becomes `north` after episode 200. Therefore, in order to learn the new policy the ϵ-greedy strategy should go repeatedly against the optimal policy for 8 times in a row, which is very unlikely. The action `south`, instead, is removed for the PRL agent by the planner, so that when it randomly moves north once, it is forced to proceed towards the goal (cf. Figure 2). This goal-directed exploration allows the PRL agent to discover the new policy even if exploring with a simple strategy such as ϵ-greedy.

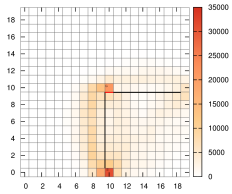
The cumulative reward achieved by the RL agent is considerably lower than that of the other agents, even in the first 200 episodes when its behavior is optimal. The unrestricted exploration can take this agent rather far from the optimal policy, both on the left and on the right-hand side of the grid. We have seen how a sensibly restricted exploration can instead obtain a better performance. Note that the path the PRL agent executed most often is not optimal (even though it is very close). This discrepancy is due to the function



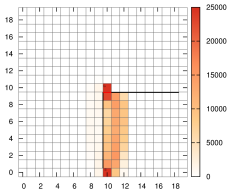
(a) RL agent's initial exploration.



(b) PRL agent's initial exploration.



(c) RL agent's final policy.



(d) PRL agent's final policy.

Figure 6: States traversed during the first and last 50 episodes by the RL and PRL agents.

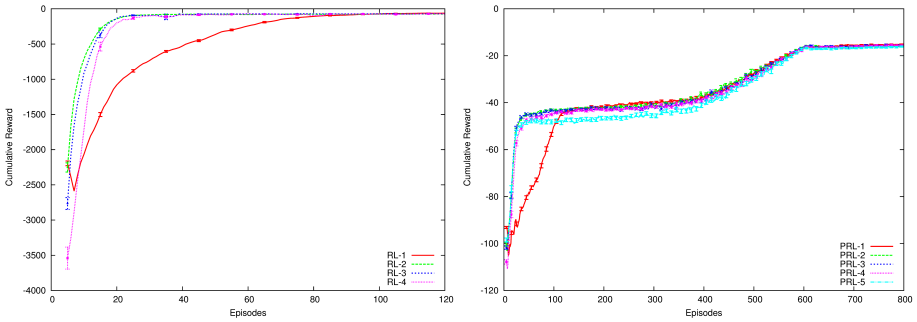
approximator employed, which cannot model the value function near the wall correctly. Since it generalizes over positions that are 2 and 5 squares apart, the value of the cells on one side and on the other side of the wall are continuously mixed. Every time an exploratory action on the left-hand side of the wall is taken, the low reward experienced drags down the estimate on the other side of the wall too. On the other hand, executing the optimal policy brings up the value not only of the states on the right-hand side of the wall, but on both sides. As an effect, when the agent explores on the left-hand side, the path along the wall looks best, even though it has the most negative reward. We will see that this is only a consequence of the function approximator, and does not happen if the agent learns with a tabular representation.

The restriction of the MDP provided by our method has three desirable outcomes, as demonstrated in this experiment: it prevents the agent from exploring the whole environment, it makes the exploration goal-directed, allowing the agent to discover new optimal policies focusing on the paths computed by the planner, and it reduces the gap between the value of the optimal policy and the reward actually obtained even while exploring.

Knowledge Representations for Learning. We analyzed further the role of the representation on learning on both the full and the reduced MDP. We ran the same experiment with four more representations for both the RL and PRL agents, whose parameters are shown in Table 1. The first representation is tabular and is therefore the baseline in terms of quality of the solution learned. The second representation is the one employed by the agents discussed above. The other three representations have been chosen for their incrementally smaller memory size, which allows us to assess the effect of representational power over the performance of the agents.

#	Layers	Tilings	Resolution	Memory
1	1	1	1	1200
2	2	16/4	5/2	512
3	1	16	5	512
4	1	8	5	256
5	1	8	10	64

Table 1: Parameters of the function approximators used for the reinforcement learning agents.



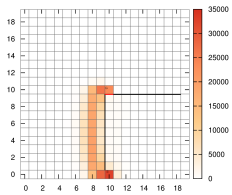
(a) The RL agents.

(b) The PRL agents.

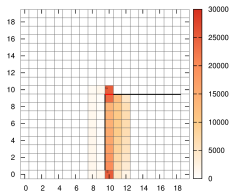
Figure 7: Reward accumulated by the learning agents with different knowledge representations

The resulting cumulative rewards are shown in Figure 7. The PRL agent is generally not affected by the degradation in the quality of the knowledge representation. The agent PRL-1, which learns with the tabular representation, is naturally slower than the ones using function approximation. The quality of the final solution, however, does not seem to be affected by approximation. The same holds for the RL agent, although in this case there is a significant difference in the initial convergence speed (Figure 7a), which degrades with the accuracy of the approximator, to the point that the RL agent is not able to learn the task with the smallest representation (number 5). The PRL agent, instead, can learn the task with this representation as well. Hence, the reduced MDP in which the PRL agent learns, besides the advantages about the exploration discussed above, also eases the requirements on the accuracy of the value function, proving simpler to learn.

Lastly, the policies learned by the RL and PRL agents with tabular value function representation are shown in Figure 8, for comparison with what is learned with function approximation (Figure 6). This figure confirms that the slightly suboptimal policy learned by the PRL agent with function approximation is indeed due to its knowledge representation, since the agent that uses a tabular value function can learn the optimal policy. All these heat maps (including the ones above) also show the typical exploration pattern of ϵ-greedy, which accounts for the difference between the performance of the learning agents and of the P agent in Figure 4.



(a) The RL agent.



(b) The PRL agent.

Figure 8: States traversed during the last 50 episodes by the RL and PRL agent with tabular representation.

The need to explore the whole domain is what often limits the feasibility of RL with no prior knowledge in practice. We will further demonstrate this claim with the experiments on a robotic domain. The grid-world domain was designed to illustrate the main characteristics of DARLING, and established them with clean results. With the next sections we apply our method to real-world scenarios.

5.2.2. Robot Simulation

With this domain we move towards more realistic scenarios: a mobile robot navigating through a building. We used the simulator Gazebo[2], and the Robot Operating System (ROS)[3]. The code that controls the robot in this simulation is the same that controls our real robot (cf. Section 5.2.3). With this simulation we demonstrate how the robot can adapt, through reinforcement learning, to specific changes in the environment, while being able to carry out the tasks efficiently thanks to planning.

Figure 9 shows the simulation environment, with the robot in its initial location, next to `b1`. The task is to go to the room marked with the red circle. The robot has the actions `approachDoor(D)`, `openDoor(D)`, and `goThroughDoor(D)` for each door in the map. The action `approachDoor` can be executed only from the locations where the door is directly accessible, that is without having to go through another door. For instance, in the room where the robot starts it can only access the doors marked with `b1`, `b2`, and `b3`. The action `goThroughDoor` can only be executed when in front of the door, and if the door is open. The action `openDoor` can be executed only when both the robot is in front of the door, and the door is closed. The immediate reward we defined for this domain is:

$$r(s, a, s') = -t(s, a, s')$$

where $t(s, a, s')$ is the time in seconds it took to execute action a from s to s'. Therefore, the robot attempts to minimize the total time it takes to reach the target location.

[2]http://gazebosim.org/
[3]http://www.ros.org/

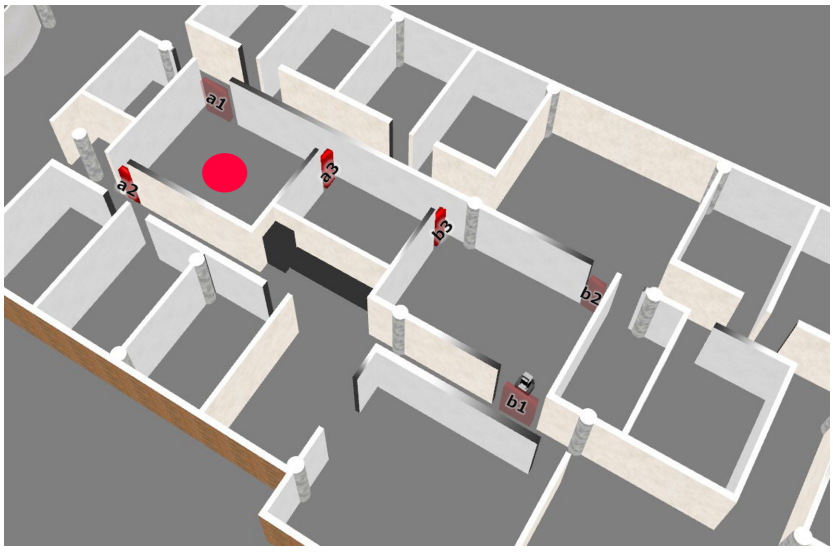


Figure 9: The simulation environment.

Knowledge Representation. The state space $\mathcal{S} = \{\langle x, y, \boldsymbol{d}\rangle\}$, where, as in the grid-world domain, x and y are the coordinates of the agents in the map, and $\boldsymbol{d}$ is a vector with the knowledge about all the doors. State observations are mapped by a specific software module called the *logical navigator* into ASP fluents, implementing the function o. At the ASP level, the state is represented with the following fluents, where `I` is the time step: `at(L,I)` which means that the robot is at location `L`, `open(D,I)`, which means that the door `D` is open, `beside(D,I)`, which means that the robot is beside door `D`, and `facing(D,I)`, which means the the robot is facing door `D`. The logical navigator looks for the current coordinates in the map, and determines the location the robot is in, and whether it is near a door or facing it. After each episode the robot goes back to the initial location and resets its knowledge base, ensuring that the initial state of its knowledge is always the same. Since the actions of the robot only allow it to move from door to door, the initial state has to be in front of a door. In particular, the initial state is: `at(roomB,0)`, `beside(b1,0)`, `facing(b1,0)`, `open(b1,0)`. Starting in front of a door, the robot can also sense whether it is open or closed.

The actions available to the robot are very high-level, and therefore the changes they make in the environment can be sufficiently described (in particular, with the Markov property) at the same symbolic level as the model. For this reason, on the robot, we did not use function approximation, but the agent learns a tabular value function on the same state space as that of the planner.

Dealing with Doors. The robot does not have any physical means to open doors. The action `openDoor` shows a window on the robot's screen with a request

for a nearby person to kindly open the door for it. While this action works in the real world (although quite unpredictably), there is no one to open the doors for the robot in the simulator. After 30 seconds the action fails, and the robot can replan. The P agent, which executes one of the shortest plans, when replanning in front of a door, always chooses to try to open that door again. This means that in this situation the robot is stuck in front of a closed door indefinitely, even if there are other ways to reach the target location. This discrepancy between the model and the reality is emblematic of the difficulty of representing a certain domain in any given formalism. For every formalism there is a trade-off between expressive power and computational complexity, which the designer must evaluate. More complex formalisms, such as probabilistic ones, may represent this domain more accurately, but at the cost of more expensive planning.

The RL and PRL agents, on the other hand, for each attempt at opening a door receive a reward of -30 (the time they sit in front of the door waiting for someone to open it). Eventually the expected reward for opening the door becomes lower than that of taking some other action, and they give up on the door. This adaptive behavior allows both agents to avoid being stuck in this domain, where doors cannot be opened.

Non-Stationarity. In this experiment we had the environment change over time in a controlled way. We designed two types of changes: (1) the current optimal path becomes worse, and (2) the value of the current path does not change, but a better solution opens somewhere else. While the agents can react to the first type of change, since they can perceive it, they can adapt to the second type only through continuously exploring the environment. The agents start in the situation depicted in Figure 9, with the doors `a1`, `b1`, and `b2` open. After 200 episodes `a1` closes and `a2` opens, starting phase 2. After 200 more episodes phase 3 begins, during which `a2` remains open, `a1` remains closed, while both `a3` and `b3` are opened. We chose the duration of the phases to be long enough to be sure that the learning algorithm has converged. The parameters of the learning algorithm are $\alpha = 0.2$, and $\epsilon = 0.2$. The episode times out after 5 minutes, and the agents go back to the initial position and start over.

Results. The P agent always chooses with uniform probability one of the two shortest plans: (1) `goThroughDoor(b1)`, `approachDoor(a1)`, `openDoor(a1)`, `goThroughDoor(a1)`; (2) `goThroughDoor(b1)`, `approachDoor(a2)`, `openDoor(a2)`, `goThroughDoor(a2)`. The door `a1` is actually open, but the robot does not know it in the initial state, and assumes it to be closed. In the first phase, the first plan succeeds in about 93 seconds, while the second plan has the robot wait for door `a2` to open until the timeout. In the second and third phases the first behavior fails, while the second one succeeds in about 64 seconds. Figure 10 shows the results of this experiment, where we plotted the task completion time (lower is better) over the episode number. For the P agent, we only plotted the average time to complete the better of the two plans in each phase. The average performance of the agent would be $G = 0.5t + 0.5T$ where t is the time

to complete the optimal plan (shown in the figure) and T is the timeout. The average performance could be made arbitrarily worse by increasing the timeout.

For the RL agent, we ran a 48-hour simulation in real time, which corresponds to about 2.5 as much simulated time, during which the RL agent could not complete phase 1. The RL agent in any of the corridors can approach more than 20 doors, which makes a thorough exploration infeasible.

We ran 10 simulations with the PRL agent (each one taking about 14 hours in real time), and the results are shown in Figure 10, averaged over a sliding window of 10 episodes. At the very beginning the PRL agent is exploring the

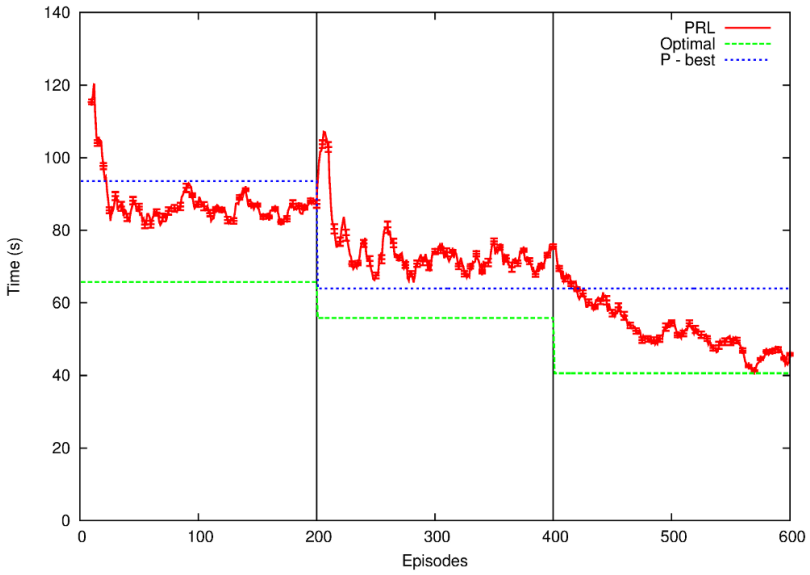


Figure 10: Time per episode in the simulation environment.

part of the environment determined by the partial policy, until it converges to the optimal solution of traversing first `b2` and then `a1`. After episode 200 the doors in the target room change status, and the time the agent takes to complete the task increases. Shortly after, the agent converges to a better solution, and stays there until the beginning of phase three. After episode 400 a better path becomes available going through `b3` and `a3`. The adaptation of the agent to this change, however, is less sudden than to the previous one, since the change has to be discovered through exploration, and is not apparent in the reward. The slope towards the new optimal solution in this phase is therefore less steep, because in the different trials the agent discovered the new solution after different amounts of time. In all the trials, however, by the end of the last phase it had converged to the new optimal path. We verified that the agent discovered the optimal solution for each phase in every trial. The difference between the performance of the agent and the expected reward of the optimal solution is only due to

exploration.

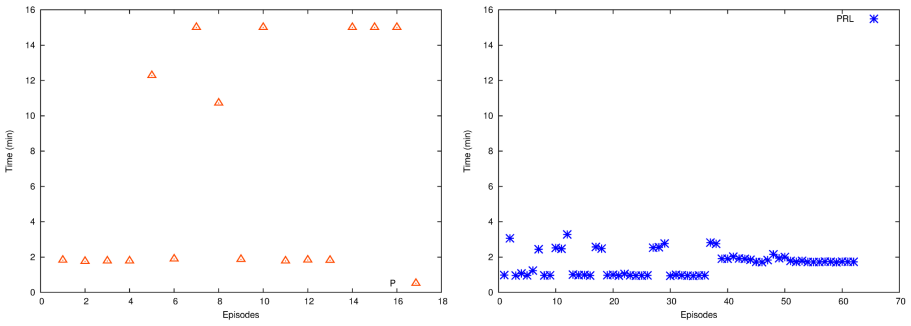
5.2.3. Service Robot

The last experiment was carried out on a BWIBot, one of the robots built at UT Austin for the Building-Wide Intelligence project, shown if Figure 11. The robot executed three tasks for about two weeks, for between 3 and 5 hours a day: one navigation task, and two Human-Robot Interaction (HRI) tasks. The navigation task is the real-word counterpart of the simulated experiment. The building modeled in the simulator is, indeed, the same where the real robot experiment also took place. In this case, however, we were not able to control the presence of people to help the robot, and the state of all doors.

Figure 11: UT Austin BWIBots.

The real robot has additional actions, with respect to the simulated robot, which allow it to interact with people. The action `searchRoom(P,R)` can only be executed when the robot is in room `R`, and asks (at the same time on the screen, and through speech) whether person `P` is in room `R`. The robot does not do speech recognition, however, so the person has to answer by going to the robot's screen and clicking on a button. The robot waits for 60 seconds, after which it assumes the person is not there. The action `askperson(P1,P2)` can be executed when the robot is in the same room as person `P1`, and asks to `P1` (again at the same time on the screen and through speech) if they know where person `P2` is. The person can answer giving the robot `P2`'s location, saying that they do not know, or saying that `P2` is not in the building.

The state is represented with the additional fluents `inroom(P,R,I)`, meaning that person `P` is in room `R`, and `inbuilding(P,I)`, meaning that person `P` is in



(a) The P agent. (b) The PRL agent.

Figure 12: Completion time for the navigation task.

the building. The robot knows 7 people in the group, 4 of whom are faculty or staff, and have offices, and 3 of whom are students. For each faculty or staff member the robot knows what office is assigned to them. Faculty and staff members are assumed to be either in their office or in the lab, while students can only be in the lab.

The learning parameters for this experiment are $\alpha = 0.2$ and $\epsilon = 0.2$. If the robot does not achieve the goal in 15 minutes, the episode terminates, the robot goes back to the initial location, and resets its knowledge base. In addition, to guarantee the responsiveness of the robot, planning is always interrupted after 10 seconds, which may limit the set of plans used to induce the MDP for which the PRL agent learns a policy. The PRL agent has been evaluated for 10 days, while the P agent for 3 days, which is sufficient to assess its behavior because it does not adapt. The RL agent has not been tested on this domain, since it proved impractical in the simulated experiment, and the real tasks have an even larger state space and higher branching factor.

Figure 12 shows the completion times for the navigation task. The robot always started from the door corresponding to `b1`, but in the real experiment it began from the corridor instead of inside the room. From there, it can plan to approach either `a1` or `a2`. If `a2` is open, going through it is the shortest path, and takes about 1 minute, while going through `a1` takes between 1.5 and 2 minutes from the initial location.

The P agent (12a) chooses one of the two available plans at random at the beginning, and when the agent chooses a door that is closed, and nobody passing by opens it, the robot keeps replanning to open that door until the time out. The PRL agent (12b), on the other hand, never reached the time out. Furthermore, for the first 36 episodes it found door `a2` open often enough to go there first, while after that it switched to the other door. Since in this task the initial choice is only between two actions, it is easy to visualize the estimated long-term cost from the initial state, which is on what the choice of the first door to approach is based. Figure 13 shows the action value function of the PRL agent, for the two initial actions `approachDoor(a1)` and `approachDoor(a2)`.

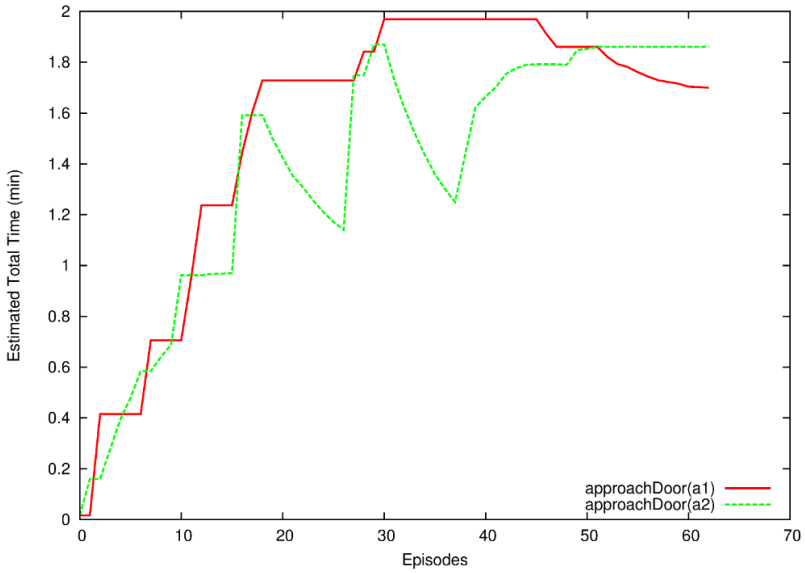
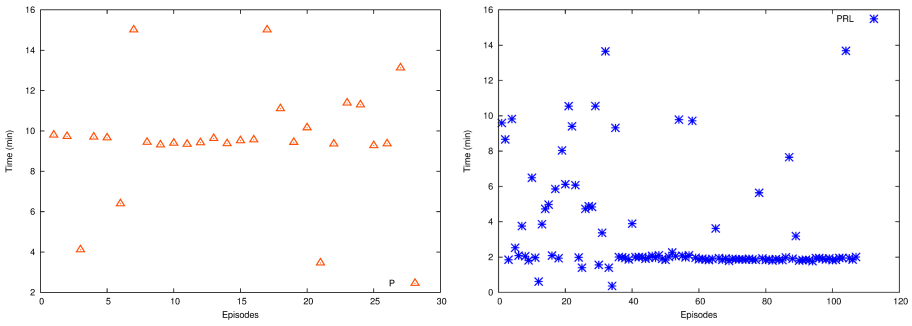


Figure 13: Action values of the PRL agent in the initial state, during the navigation task.

The value of both actions starts at an optimistic estimate of 1 second, and for the first about 20 episodes the robot took both actions in turn adjusting their estimate. In an episode when an action is not taken, its value remains constant. From episode 19 going to door `a2` looks like the better option, and the robot keeps taking that action, with the estimate approaching the total time. At episode 27 the robot must have found the door closed, since the estimate went suddenly up, reaching the worse action. The actions have been executed in turns again for a few episodes, until `a2` proved still to be the better option. The estimate value of action `a2` started rising again at episode 37, when the robot began to find `a2` closed. The estimate of going to `a2` first reached the estimate for going directly to `a1`, and the robot switched to the new better action, adapting to the environment.

The HRI tasks have the goal of finding two different people, to whom we will refer as `personA`, and `personB`. The robot has to either confirm the location of a person, or confirm that the person is not in the building. The robot can do so by either having `searchroom` succeed, or by being told through an `askperson` action. The results of the HRI experiment when the robot was looking for `personA` are shown in Figure 14. The time the robot takes to complete this task depends on a number of factors, all very difficult to model: which rooms the robot manages to enter more often, whether or not the people that the robot knows about can be found and asked about `personA`, and how often they are willing to answer the robot. We noticed that the students, for instance, were eager to answer the robot at the beginning, but soon started to ignore it. Since



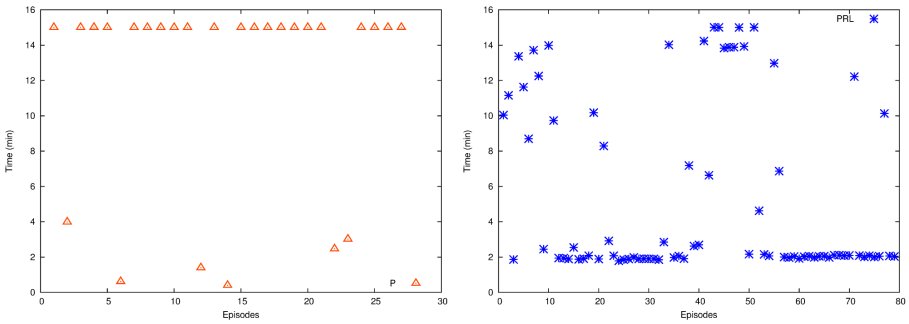
(a) The P agent. (b) The PRL agent.

Figure 14: Completion time for HRI task with `personA`.

the robot started in front of the lab, when it entered the lab, found a person, and that person answered, it could quickly complete the task. Otherwise, it had to go to different offices to search for `personA`, or people who might know about him. During most of the episodes, `personA` was usually sitting in his office. After about 40 episodes, the robot running the PRL agent converged to the solution that goes directly to `personA`'s office, and searches in there. This policy completes the task quite reliably in about 2 minutes. Even after having converged to the optimal policy, the robot chooses a random action with probability $\epsilon = 0.2$, which accounts for the episodes in which the robot took a longer time to complete the task. As we have shown on the simulated domain, keeping the exploration active is necessary to track changes in the environment.

During the trials where the robot ran the P agent, it reached the timeout twice, and on average it takes a little less than 10 minutes to complete the task. The P agent always enters the lab and tries to ask the people in there, never adapting to whether or not they are willing to answer. After that, it replans its way towards other offices, and eventually finds `personA` in his office.

The results of the second HRI task, in which the robot looks for `personB`, are shown in Figure 15. In this case, `personB` was often not in the office, or with his door closed, so when the P agent plans to go directly to his office, it ends up waiting in front of the door until the episode times out. The PRL agent, on the other hand, learned an interesting behavior, to which it converged by episode 60: it goes directly to `personA`'s office (since he was reliably there!), and asks about `personB`. This behavior allows the PRL agent to complete the task in the same amount of time as the previous task. In this task, however, differently from the previous one, the P agent fails much more often, and is unable to complete the task in the majority of the episodes.



(a) The P agent. (b) The PRL agent.

Figure 15: Completion time for HRI task with `personB`.

6. Conclusions

Automated planning and reinforcement learning are two different paradigms for solving complex decision making problems. Both paradigms face challenges in high-level decision making on large, unpredictable, and non-stationary domains, especially in robotics. Their characteristics, however, admit an integration that can benefit from their stronger features, and overcome some of their respective weaknesses. We proposed a method, DARLING, to combine automated planning and reinforcement learning, which extends the applicability of both paradigms to scenarios where each one separately would not be sufficient.

We analyzed the performance of DARLING in domains in which a discretization of the state space appeared natural, which allowed us to use ASP. In high-dimensional continuous domains it may not be possible to find such a discretization, and a similar argument holds for the discretization of continuous actions. This is a limitation we recognize in the approach as presented in this article, and significant adjustments may be necessary to DARLING, and the planning formalism employed, to deal with such domains. The total number of plans computed by the ASP solver in general increases exponentially in the length of the shortest plan. As a consequence, in the real-world domains the number of plans was often limited by planning time, rather than plan length. The deadline gives the agent no control on which plans are included and which ones are discarded. A possible interesting development is the use of a set of *diverse plans* [29] (also in ASP [9]) to focus the search, and on which to expand during execution. Lastly, we showed in Section 4.4 how the metric used for the threshold does not have to be related with the actual cost of actions. However, if plan length, which is the metric we used, does not permit to define a satisfactory threshold function, the designer will have to define an ad-hoc one, and to employ a planner that minimizes it. Sampling-based planning, which in recent years has scaled up to the most challenging domains, could be employed in that case.

Within the range of applicability of DARLING, we showed how, on the planning side, the ability to learn lightens the reliance on the accuracy of the model,

and allows the agent to adapt to the environment. On the reinforcement learning side, the rationality of the automated reasoner, and the previous knowledge in the model, allow the agent to exclude certain actions without the need to try them. We showed in different experiments how this integration modifies the region of the environment explored by the agent, how it allows the agent to adapt to a non-stationary environment, and how it enabled a service robot to carry out different tasks over an extended period of time, where planning would often fail and reinforcement learning could not complete any of the tasks.

Acknowledgements

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. LARG research is supported in part by NSF (CNS-1330072, CNS-1305287), ONR (21C184-01), and AFOSR (FA9550-14-1-0087). Peter Stone serves on the Board of Directors of Cogitai, Inc. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research.

Appendix: Modeling the Grid World in ASP

In this section we illustrate how to represent a model in ASP for our Grid World domain. The state of the domain is given by the position of the agent, and the position of the obstacles (since the door can change). We represent the states through the fluents `pos(X,Y,I)`, meaning that at time step `I` the position of the agent is $\langle X, Y \rangle$ and `obst(X,Y,D,I)` meaning that there is an obstacle at $\langle X, Y \rangle$ in the direction D (one of the four cardinal directions). The description of any domain in ASP is composed of a set of laws of different types: dynamic laws, static laws, constraints, and choice rules.

Dynamic laws describe the effect of actions, in our example :

```
pos(X,Y+1,I+1) :- north(I), pos(X,Y,I), I=0..n-1.
```

encodes the effect of the action `north`, which increments the Y coordinate of the position fluent. Constraints can be used to describe pre-conditions as in:

```
:- north(I), pos(X,Y,I), obst(X,Y,no,I), I=0..n.
```

which means: it is not possible that the agent executes action north, it is in position $\langle X, Y \rangle$, and there is an obstacle north of $\langle X, Y \rangle$, at time step I. We use the symbols `no`, `s`, `e`, and `w` to indicate the four cardinal directions, while n is a variable that is required by the machinery, and will be incremented by the solver to increase the maximum time step.

A choice rule:

```
1{north(I),east(I),west(I),south(I)}1 :- I=0..n-1.
```

allows the reasoner to generate exactly one action per time step.

The way the fluents unaffected by the executed action change is specified by default dynamic laws such as:

```
pos(X,Y,I+1) :- pos(X,Y,I), not -pos(X,Y,I+1), I=0..n-1.
```

which is an inertial law for the fluent `pos`, that is, it states that `pos(X,Y,I)` carries over to the next time step unless proven otherwise. All defaults are expressed through negation as failure. For this law to work as expected we also need to define how `-pos` can be true. When the agent is at some position, it is not anywhere else:

```
#const max_x=19.
#const max_y=19.
-pos(X,Y,I) :- pos(Z,K,I), Z != X, X=0..max_x, Y=0..max_y.
-pos(X,Y,I) :- pos(Z,K,I), K != Y, X=0..max_x, Y=0..max_y.
```

Lastly, we define the obstacles, both for the border of the grid:

```
direction(no). direction(s).
direction(e). direction(w).

obst(X,0,s,I) :- X=0..max_x, I=0..n.
obst(0,Y,w,I) :- Y=0..max_y, I=0..n.
obst(X,max_y,no,I) :- X=0..max_x, I=0..n.
obst(max_x,Y,e,I) :- Y=0..max_y, I=0..n.
```

and inside the grid:

```
obst(9,Y,e,I) :- Y=1..9, I=0..n.
obst(X,9,no,I) :- X=11..18, I=0..n.
```

Obstacle fluents are symmetric, if there is an obstacle in $\langle X, Y\rangle$ going north, there is also an obstacle in $\langle X, Y+1\rangle$ going south and so on:

```
obst(X,Y+1,s,I) :- obst(X,Y,no,I).
obst(X,Y-1,no,I) :- obst(X,Y,s,I).
obst(X+1,Y,w,I) :- obst(X,Y,e,I).
obst(X-1,Y,e,I) :- obst(X,Y,w,I).
```

As for the fluent `pos`, `obst` is inertial, that is it stays unmodified unless explicitly changed:

```
obst(X,Y,Z,I+1) :- obst(X,Y,Z,I), not -obst(X,Y,Z,I+1),  I=0..n-1.
```

By default, there is no obstacle between two states:

```
-obst(X,Y,Z,I) :- not obst(X,Y,Z,I), direction(Z), ...
    I=0..n, X=0..max_x, Y=0..max_y.
```

As shown by this example, thanks to defaults and inertial laws, the domain can be compactly represented in ASP.

[1] Pieter Abbeel, Morgan Quigley, and Andrew Ng. Using inaccurate models in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, volume 148, pages 1–8. ACM, 2006.

[2] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[3] George EP Box. Robustness in the strategy of scientific model building. *Robustness in statistics*, 1:201–236, 1979.

[4] Ronen I Brafman and Moshe Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3(2):213–231, 2003.

[5] Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009.

[6] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1):35–74, 1995.

[7] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1):7–52, 2001.

[8] Kyriakos Efthymiadis and Daniel Kudenko. Using plan-based reward shaping to learn strategies in starcraft: Broodwar. In *Proceedings of IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013.

[9] Thomas Eiter, Esra Erdem, Halit Erdogan, and Michael Fink. Finding similar/diverse solutions in answer set programming. *Theory and Practice of Logic Programming*, 13:303–359, 5 2013.

[10] Alan Fern, Roni Khardon, and Prasad Tadepalli. The first learning track of the international planning competition. *Machine Learning*, 84(1):81–107, 2011.

[11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011.

[12] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[13] Malik Ghallab, Dana Nau, and Paolo Traverso. The actor[U+02BC]s view of automated planning and acting: A position paper. *Artificial Intelligence*, 208:1 – 17, 2014.

[14] Marek Grzes and Daniel Kudenko. Plan-based reward shaping for reinforcement learning. In *Proceedings of the 4th International IEEE Conference on Intelligent Systems (IS)*, volume 2, pages 10–22, 2008.

[15] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[16] Matteo Leonetti, Luca Iocchi, and Fabio Patrizi. Automatic generation and learning of finite-state controllers. In *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, pages 135–144. Springer, 2012.

[17] Vladimir Lifschitz. Answer set planning. In *Logic Programming and Nonmonotonic Reasoning*, pages 373–374. Springer, 1999.

[18] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1594–1597. AAAI Press, 2008.

[19] Timothy Mann and Yoonsuck Choe. Scaling up reinforcement learning through targeted exploration. In *AAAI Conference on Artificial Intelligence*, pages 435–440, 2011.

[20] Allen Newell and Herbert A Simon. *GPS, a program that simulates human thought*. Defense Technical Information Center, 1961.

[21] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[22] Nils J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[23] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.

[24] O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[25] Jervis Pinto and Alan Fern. Learning partial policies to speedup mdp tree search. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 672–681, 2014.

[26] Malcolm RK Ryan. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *Proceedings of the International Conference of Machine Learning (ICML)*, volume 2, pages 522–529, 2002.

[27] Malcolm RK Ryan and Mark D Pendrith. Rl-tops: An architecture for modularity and re-use in reinforcement learning. In *Proceedings of the International Conference of Machine Learning (ICML)*, pages 481–487, 1998.

[28] Harm V. Seijen and Rich Sutton. True online td(lambda). In *Proceedings of the 31st International Conference on Machine Learning (ICML)*, pages 692–700, 2014.

[29] Biplav Srivastava, Tuan Anh Nguyen, Alfonso Gerevini, Subbarao Kambhampati, Minh Binh Do, and Ivan Serina. Domain independent approaches for finding diverse plans. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2016–2022, 2007.

[30] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[31] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[32] Martijn van Otterlo. A survey of reinforcement learning in relational domains. Technical Report TR-CTIT-05-31, Enschede, the Netherlands, 2005.