In *Proceedings of the IJCAI Workshop on Autonomous Mobile Service Robots* , New York City, NY, USA
July 2016

# State Aggregation through Reasoning in Answer Set Programming

**Ginevra Gaudioso, Matteo Leonetti, and Peter Stone**
Department of Computer Science
University of Texas at Austin
ginevra.gaudioso@utexas.edu, {matteo,pstone}@cs.utexas.edu

## Abstract

For service robots gathering increasing amounts of information, the ability to realize which bits are relevant and which are not for each task is going to be crucial. Abstraction is, indeed, a fundamental characteristic of human intelligence, while it is still a challenge for AI. Abstraction through machine learning can inevitably only work in hindsight: the agent can infer whether some information was pertinent from experience. However, service robots are required to be functional and effective quickly, and their users often cannot let the robot explore the environment long enough. We propose a method to perform state aggregation through reasoning in answer set programming, which allows the robot to determine if a piece of information is irrelevant for the task at hand before taking the first action. We demonstrate our method on a simulated mobile service robot, carrying out tasks in an office environment.

## 1 Introduction

Service robots will have to perform a variety of tasks for extended periods of time, inhabiting their environment permanently. They are subject to continuously accumulating knowledge, which can quickly become overwhelming. Decision making requires robots to process such knowledge repeatedly, with the additional challenge, brought about by sharing the environment with humans, of having to do so with a responsiveness that does not make the users impatient.

Being able to realize which pieces of information are relevant for which task will become an indispensable skill. This ability of *abstracting* irrelevant details is a formidable characteristics of human intelligence, which is still a challenge for AI. A person learning the shortest path between two rooms in a building would have no problem realizing that the weather outside has no bearing on his task. He would not need to learn it from data, trying a certain path with sunshine and with rain to verify it takes the same time. He would just be able to realize it by reasoning. This human capability is what we want to achieve by automated reasoning.

In this paper, we propose a system which leverages automated reasoning in Answer Set Programming (ASP) to determine a set of possible courses of action for the robot, and the respective relevant information. Reinforcement learning is then performed to learn the best behavior for the task, learning decisions that depend only on the knowledge that the ASP reasoner has deemed pertinent.

By reasoning on a model, the agent can direct the execution towards the goal, without the need to explore the whole environment. Through reinforcement learning, the agent can adjust to the environment, adapting in the (very likely) case of inaccuracies of the model. By learning alone, however, the only way to realize whether some features of the perceived state are irrelevant for the optimal behavior is in hindsight: after trying the same actions multiple times, statistical tests can reveal that certain states can indeed be abstracted. In certain cases as our example above, on the other hand, we expect that reasoning on a model of the environment should provide such an abstraction even before acting. For this reason, we propose a combination of automated reasoning in ASP and reinforcement learning, to benefit from both the forward view of reasoning and the backward view of learning.

## 2 Background

### 2.1 Answer Set Programming

Answer Set Programming is a form of declarative programming, based on the stable model semantics of logic programming [Lifschitz, 2008]. Its 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 our 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 implication sign separates the *head* (on the left), from the *body* (on the right) 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 [1988].

The symbol *not* is often referred to as *negation as failure*. The classical negation of an atom `p` is denoted with `-p`, and is another atom, with the 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
```

If the body is verified by the answer set, then zero or any number of atoms in $\{\mathtt{p}, \ldots, \mathtt{q}\}$ may be in the answer set. Two other special cases of rules are *facts* and *constraints*. A *fact* is a rule in which the body is missing. A *constraint* is a rule in which the head is missing. For instance, "`:- q`" means that `q` cannot belong to any answer set.

To compute the answer sets of a given program, we used the answer set solver `clingo` [Gebser *et al.*, 2011].

## 2.2 Planning in Answer Set Programming

We want to represent a dynamical system $D_m = \langle \mathcal{S}, \mathcal{A}, f_m \rangle$ in Answer Set Programming. The set $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, and $f_m : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ is the (potentially non-deterministic) transition function.

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*. Given a finite set of fluents, $\mathcal{F}$, the set of states $\mathcal{S} = 2^{\mathcal{F}}$ is the set of all possible truth assignments (positive or negative) to the fluents in $\mathcal{F}$. In ASP, however, a fluent may have *three* values: appear in an answer set, appear negated, or not appear at all. Therefore, answer sets can be partial assignments to the fluents, where some fluents result unspecified. Such answer sets may correspond to more than one state (the ones obtained by assigning *true* or *false* to the missing fluents in all possible ways), and we will refer to them as *belief* states. Let $\mathcal{B} = 3^{\mathcal{F}}$ be the set of belief states over the fluents in $\mathcal{F}$. Let $c : \mathcal{B} \to 2^{\mathcal{S}}$ be a completing function which for any belief state $b$ returns the set of states obtained by assigning to the fluents in $\mathcal{F} \setminus b$ truth values in all possible ways. We can establish a partial ordering over belief states, determining that a belief state $b$ is *more general* than a belief state $b'$ iff $c(b) \supseteq c(b')$, denoted with $b \succeq b'$.

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.

The pre-condition of actions is represented with constraints. For instance:

```
:- openDoor(D,I), not facing(D,I)
```

means that it is not possible to execute action `openDoor(D)` if `facing(D)` cannot be proven at time step `I`.

The effect of actions is represented with rules such as:

```
open(D,I+1):- openDoor(D,I)
```

which means that executing the action `openDoor(D)` at time step `I` causes the door to be open at time step `I+1`.

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 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 $\mathcal{G}$.

## 2.3 Optimization in Answer Set Programming

The answer set solver `clingo` supports optimization statements, which allow the programmer to obtain not just any answer set but an *optimal* one. We take advantage of this feature, since we want to compute the *minimum* set of fluents necessary to estimate the cost of a plan. An optimization statement can be specified as follows:

```
opt { L_1, ..., L_n }
```

where `opt` can be either `#minimize` or `#maximize`, and the symbols $\mathtt{L}_n$ are literals. As a consequence of the presence of this statement, the answer set solver will return an answer set which has either the minimum or the maximum possible number of literals specified in the statement.

Answer set programming allows us to write very compact models, and the answer set solver `clingo` can return not just one plan but all plans that fulfill certain constraints, which will be necessary for the definition of our method. Furthermore, the ability to optimize over the answer sets allows us to define the *minimum* solution as the more abstract, enabling the reasoning at the foundation of state aggregation. A drawback of automated reasoning is that decisions originate from the model only, and any discrepancies between the environment and the model may lead to a failure. We overcome this weakness through reinforcement learning, grounding decisions also on data from experience.

## 2.4 Markov Decision Processes

We introduce briefly in this section the notation we use for Markov Decision Processes, referring to one of the many more specialized texts for a complete discussion [Sutton and Barto, 1998]. We denote a Markov Decision Process as $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $f$ is the transition function and $r$ is the reward function. We denote with $\mathcal{A}(s)$ the actions available in state $s$. The behavior of the agent is represented as a function $\pi : \mathcal{S} \to \mathcal{A}$ called a *(stationary deterministic) policy*, which returns the action to execute in each state.

A number of methods exist which return an optimal policy by computing a *value* function:

$$q_\pi(s,a) = \sum_{s'} f(s,a,s')(r(s,a,s') + \gamma q(s', \pi(s'))), \quad (1)$$

where $0 < \gamma \leq 1$ is the *discount factor*. The value function computes the expected return for taking action $a$ in $s$ and following $\pi$ thereafter. A policy $\pi^*$ is *optimal* iff $q_{\pi^*}(s,a) \geq q_\pi(s,a)$, for every other policy $\pi$, and $\forall s \in \mathcal{S}, a \in \mathcal{A}$.

Value functions often cannot be represented exactly but have to be approximated. Function approximation also allows the agent to generalize between *similar* states (where the similarity depends on the function approximation involved).

A popular function approximator, for which implementations are widely available, is Tile Coding with hashing [Sutton and Barto, 1998]. 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 possible to have multiple tilings slightly shifted from each other span the state space, or at different resolutions.

We will use Sarsa($\lambda$) [Sutton and Barto, 1998] for control, estimating the value function with True Online TD($\lambda$) [Seijen and Sutton, 2014]. The exploration will be an $\epsilon$-greedy strategy. With the $\epsilon$-greedy strategy, the agent chooses the current optimal action according to $q_\pi$ with probability $1 - \epsilon$, and a random action with probability $\epsilon$.

Reinforcement learning allows the agent to learn an optimal behavior with no prior knowledge. In practice, however, this often requires an infeasible amount of experience. This issue is addressed by DARLING, a method combining planning and reinforcement learning described in the next section.

### 2.5 Domain Approximation for Reinforcement Learning

The approach presented in this paper builds on a method to generate a reduced MDP for reinforcement learning through planning, called Domain Approximation for Reinforcement LearnING (DARLING). DARLING comprises three steps: plan generation, plan filtering and merging, and reinforcement learning.

**Plan Generation** The user is required to provide an ASP model $D_m = \langle \mathcal{S}, \mathcal{A}, f_m \rangle$ of an MDP $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$. For this paper we only consider MDPs that can be modeled in ASP, therefore with discrete action and state spaces. The first step consists of computing all plans of length at most $L = \mu \cdot l$ where $l$ is the length of the shortest plans, and $\mu \geq 1$ is a parameter of the method, which determines how suboptimal a plan can be in the model in order to be considered for reinforcement learning in the environment.

**Plan Filtering and Merging** Let $\mathcal{P}$ be the set of the plans computed at the previous step. Some plans in $\mathcal{P}$ may be *redundant*, that is, they may contain a sequence of actions such that, if removed from the plan, what remains is also a plan. For instance, a plan containing a cycle is redundant. If a plan is not redundant it is said to be *minimal*. Redundant plans are filtered out of $\mathcal{P}$, and the remaining plans form the set $\Pi(L)$ of all the minimal plans of length at most $L$. The plans that belong to $\Pi(L)$ are then merged into a *partial* policy

$$\pi_L(s) = \{a | \exists p \in \Pi(L) \text{ s.t. } \langle s, a \rangle \in p\}, \ \forall s \in \mathcal{S}. \qquad (2)$$

While a policy is a function that returns an action for each state of an MDP, a partial policy is a function $\pi : \mathcal{S} \to 2^{\mathcal{A}}$ which returns the set of all the actions that belong to at least one plan. Such a function is used in the last step of DARLING to define a reduced MDP $D_r$ on which the agent can do reinforcement learning.

**Reinforcement Learning and Execution** At run time, the agent can choose only among the actions returned by the partial policy, and it estimates their value by reinforcement learning to make an informed choice. The partial policy reduces the MDP in which the agent effectively learns from $D = \langle \mathcal{S}, \mathcal{A}, f, r \rangle$ to $D_r = \langle \mathcal{S}, \mathcal{A}_r, f_r, r \rangle$, 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 $a \notin \mathcal{A}_r(s)$.

If the agent finds itself in a state for which $\pi_L$ returns no action, it can replan from that state, compute a new partial policy, and merge it with the current one, augmenting the MDP in which it learns.

## 3 Related Work

Li et al. [Li *et al.*, 2006] identify several levels of abstraction for Markov Decision Processes, from the strictest, corresponding to Bisimulation [Givan *et al.*, 2003], in which two states are aggregated only if the full one-step model corresponds exactly, to Policy Irrelevance [Jong and Stone, 2005], in which only the optimal action has to be maintained when merging two states. Our method is at an intermediate level, indicated by Li et al. as $Q^*$-Irrelevance, since it aims at preserving the optimal value function. At the same level of abstraction are the G-algorithm [Chapman and Kaelbling, 1991], and stochastic dynamic programming with factored representation [Boutilier *et al.*, 2000]. The former initially collapses every state, and later performs statistical tests to split particular states when necessary. It is a learning algorithm, which does not require a model of the MDP, but does require experience to gather enough data for the statistical tests. Conversely, our method is based on reasoning on a model, and no prior experience is required to determine which information is certainly irrelevant.

Factored representations allow for a natural description of many domains in terms of *features*. Stochastic dynamic programming on factored MDPs is a decision-theoretic method, and it works on a model of the MDP, like our method, but it requires an exact, stochastic, model. Our method, on the other hand, requires an ASP model of the environment, which does not include transition probabilities and actions costs. Related to factored representation is Relational Reinforcement Learning (RRL) [Džeroski *et al.*, 2001]. 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. Even if generalizing through first-order lifted inference, RRL methods still calculate values and policies for the full domain, while our method uses ASP (in its propositional definition) to reduce the portion of the region to explore, and apply the generalization to that region only.

Lastly, the most common method for state aggregation for RL is through function approximation. In particular, tile coding described in Section 2.4, is one of the most popular linear function approximators. For this reason, we compare our method against Tile coding in Section 5.1.

## 4 Method

As previously introduced, a weakness of reinforcement learning, inherited by DARLING, is that state aggregation has to happen in hindsight: as a consequence of the data acquired through acting. In many situations, however, it is obvious to

humans that certain features of the environment have no impact on a particular task, for instance the weather outside of a building for indoor navigation. We aim at formalizing and implementing such reasoning in ASP, so that the agent can realize that certain aspects of the environment do not matter for the task at hand, even before taking the first action.

As a motiving example, consider a service robot performing tasks in the building shown in Figure 1. The robot re-

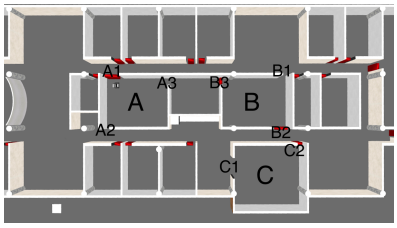



Figure 1: Simulation environment of an office building.

members whether the doors in front of it are open or closed. If the robot is in room $A$, as shown in the figure, and it has to navigate to room $B$, the state of only a few of the about 30 doors on the floor actually affects the navigation.

We are interested in determining when two states are equivalent for the purpose of computing an optimal policy in the restricted MDP $D_r$. If in two states $s$ and $s'$ the same actions are available, and each action has the same value under the optimal policy:

$$A_r(s) \equiv A_r(s') \wedge q_{\pi^*}(s,a) = q_{\pi^*}(s',a), \forall a \in A_r(s) \quad (3)$$

then $s$ and $s'$ can be *aggregated*, which we will denote with $B(s,s')$.

As introduced in Section 2.2, it is possible in ASP to reason in terms of belief states, that is, states in which some fluents are not assigned a truth value. Some belief states are more general than others, in that they correspond to larger sets of fully specified states.

For each state $s$ for which the partial policy (Eq. 2) returns a non-empty set of actions, we want to compute the most general belief state $b^*$ such that the following conditions hold:

$$s \in c(b^*) \quad (4)$$

$$B(s,s')\ \forall s,s' \in c(b^*) \quad (5)$$

$$\nexists b.\ b \succeq b^* \wedge B(s,s')\ \forall s,s' \in c(b) \quad (6)$$

that is, $b^*$ is the maximal equivalence class for the value function containing the state $s$. Such a belief state $b^*$ would not contain any of the fluents that are irrelevant for distinguishing states for the purpose of estimating the optimal value function. Therefore, it would be possible to learn the same value for all the states in $c(b^*)$, accelerating learning considerably.

Executing DARLING, the agent has already computed $\Pi(L)$. The plans are stored in a directed acyclic graph $G = \langle V, E\rangle$, where $V = \{s | \exists p \in \Pi(L) \text{ s.t. } \langle s, a_i\rangle \in p\}$, is the set of nodes which is composed of the states that are traversed by at least one plan, and $E = \{\langle s_i, a_i, s_j\rangle | \exists p \in \Pi(L) \text{ s.t. } \langle s_i, a_i\rangle, \langle s_j, a_j\rangle \in p \wedge s_j = s_{i+1}\}$ is the set of edges, labeled by the actions, which link two states if they appear one immediately after the other in at least one plan.

We require the user to specify the one-step model for estimating action costs, and we take advantage of an automated reasoner to verify which bits of information are necessary.

We add an additional requirement to the specification of actions in ASP: a pre-condition for an action $a$ is verified in a belief state $b$ if: (1) the action is executable in any state of $c(b)$, and (2) the agent has all the necessary information to correctly estimate the cost of $a$ from any state in $c(b)$. The second requirement is usually not present in planning, but it will allow the reasoner, with an appropriate query, to chain the necessary knowledge for estimating action costs.

For instance, the robot of our example has an action `approachDoor(D,I)` to approach a door $D$ at time step $I$. We use time as a metric for action costs. The pre-condition of such an action requires (1) the agent to be at a location connected to the door (to enable the action) and (2) to know the room in which the robot is, and the door it is beside. The second part of the pre-condition is necessary to be able to associate a cost with the action.

In order to compute the most general current belief state, the agent generates an ASP query for each plan available from the current state. First, the current state is located in the graph $G$. If not present, the agent can replan. Then, the graph is visited with a depth first search starting from the current state. For each plan, the agent constructs the following query:

1. For each fluent $p_i$ in the current state, add a choice rule $\{\mathtt{p}_i\}$.
2. Add an optimization statement with each fluent $p_i$ in the current state: `#minimize {p1, p2, ..., pi}`.
3. For each action `a(C,i)` with constants $C$ at time step $i$ in the current plan, add the action as a fact: `a(C,i).`
4. Add the goal.

The resulting query is similar to a planning query, since it contains the goal, but the choice rule is not on the actions, which on the contrary are specified as facts, but on the fluents of the initial state. Because of the minimization statement, only the fluents that are necessary for the plan to achieve the goal will be added. Hence, this method explicitly takes advantage of the fact that the agent knows what it is going to do, and can therefore reason about what information will be necessary down the road.

Consider the plan in the example that goes from room $A$ to room $B$ through the doors $A_1$ and $B_1$. The knowledge base of the robot could contain the state of 10 doors, still leaving about 20 more unspecified. It also certainly contains the fluent `at(roomA,0)` specifying that the robot is in room $A$, and the fluent `beside(A1,0)`, since the robot is beside door $A_1$. A minimization of the fluents of the current state would leave only the `at` and `beside` fluents, and the fluents encoding the state of doors $A_1$ and $B_2$. The state of any other door would be removed from the answer set.

The query returns the minimum answer set $AS_i$ for a particular plan $p_i$ . The minimum belief state containing the current state is then computed as $b^* = \cup AS_i$, the union of all the answer sets for every plan. The information discarded is certainly irrelevant for the plan, while it is still possible that further generalization can be done on the preserved informa-

tion. This is the most we can extract from the model, but more aggregation can be performed learning from data.

## 5 Experimental Validation

We validate our method to aggregate states through reasoning in ASP in two domains. The first domain is a grid world designed to serve as an illustration of the method. It allows us to run a large number of trials, and to compare our method with Tile Coding. The second domain is the realistic simulation of a mobile service robot introduced above.

### 5.1 Gridworld Domain

The grid-world domain designed to illustrate our method is represented in Figure 2. The state is composed of the agent's

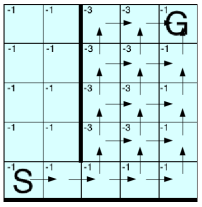



Figure 2: The Gridworld used for this experiment.

position in the grid and the grid's color. The position is represented as $\langle x, y \rangle$ coordinates, where the bottom left corner is $\langle 0, 0 \rangle$ and the top right corner is $\langle 4, 4 \rangle$. The color represents information which is irrelevant for navigation tasks in the grid. The agent moves by executing the actions `north`, `south`, `east` and `west`, which deterministically move the agent in the respective direction, unless it would take the agent out of the grid, or make it hit the wall, in which case the agent does not move. In Figure 2, the wall is the thick black line. The pre-conditions of the actions require the agent to know its position, but do not require the color.

The wall has been added to illustrate the effect of DARLING. We ran DARLING with $\mu = 1$, so that only the plans that are optimal in the model (shortest plans) are retained. The resulting reduced MDP $D_r$ contains only actions to reach the bottom cells, and the cells on the right-hand side of the wall, as shown in Figure 2. There are 15 shortest plans in this grid, but only one of them is the optimal plan in practice. Since the ASP model of the environment does not contain the reward, the optimal plan will have to be learned. The reward returned for entering each cell is also shown in Figure 2.

The agent goes from the starting position (marked with S in the figure) to the goal position (marked with G) for 100 episodes. The color of the grid was randomly assigned as part of the initial state out of a set of 100 colors.

We compare the agent performing state aggregation with our method against four agents that do state aggregation with tile coding, and one which learns in the full state space, without doing any aggregation. The agents using tile coding have as input the state vector $\langle x, y, c \rangle$, where the first two variables are the coordinates of the agent in the grid, and $c \in [0, 99]$ encodes the color of the grid. Each tile coding agent employs a group of 8 tilings with $2 \times 2$ cells on the first two variables, and different sizes on the color variable, namely: 2, 10, 50, and 100. A drawback we can immediately note about state aggregation with tile coding is that the representation has to be designed by hand. The parameters of Sarsa($\lambda$) are $\alpha = 0.1$, $\gamma = \lambda = 0.9$, $\epsilon = 0.5$. For tile coding, $\alpha$ has been normalized by the number of tilings.

The results, shown in Figure 3, are averaged over 1000 trials, and a sliding windows of 5 episodes. The plot also shows the 95% confidence intervals every 5 points. The agent implementing our method filters out the information about the color at planning time, and the input to its learning layer contains only the position of the agent. Even if learning with a tabular representation, it can outperform all of the tile coding agents. The performance of tile coding agents increases, as expected, with the generalization over the irrelevant variable. The agent learning in the full state space in tabular form is the slowest to learn, since it has to re-learn the optimal action for each state for every color of the grid.

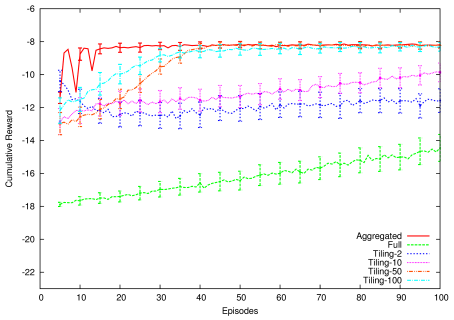



Figure 3: The results of the grid-world experiment.

In this experiment the irrelevant knowledge has been injected in the design of the environment, and can be identified easily. In the next domain we show a realistic scenario in which relevant and irrelevant fluents have to be determined state by state.

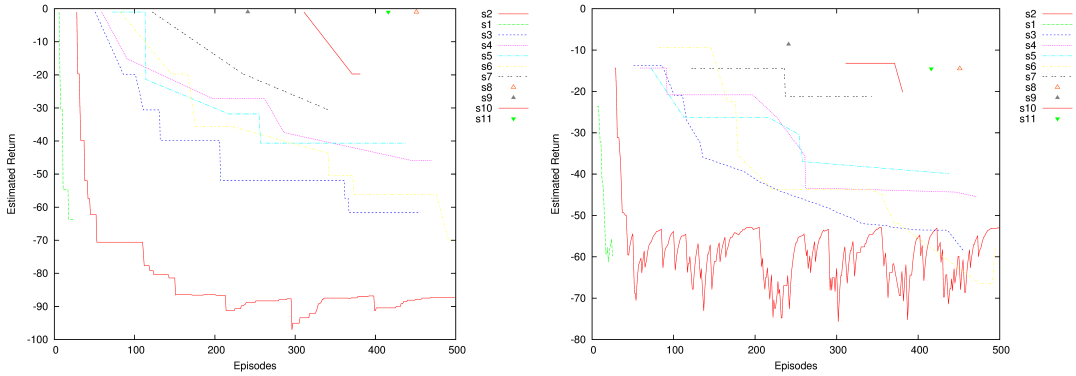
### 5.2 Robot Navigation Simulator

The second domain was introduced in Section 4 and is shown in Figure 1. The simulation is controlled by the same code that controls our robots, executed in the Robot Operating System (ROS), while the 3D simulation was built in Gazebo[1].

The state of the environment is represented by a set of fluents as follows. The `at(R,I)` fluent represents the position of the robot; `beside(D,I)` and `facing(D,I)` indicate the position of the robot with respect to door $D$; the `open(D,I)` fluents represent the state of the doors. The set of actions is: `gothrough(D,I)`; `opendoor(D,I)`; `approach(D,I)`, where $D$ is a door, with their literal meaning. The reward function 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 $s'$.

The robot had a sequence of three tasks to perform, which was repeated 500 times. Each task has the goal of reaching

[1]ROS: www.ros.org, Gazebo: gazebosim.org




(a) Task 1, action 1.

(b) Task 1, action 2.

Figure 4: The estimated return without state aggregation for the initial states corresponding to the single belief state of Figure 5

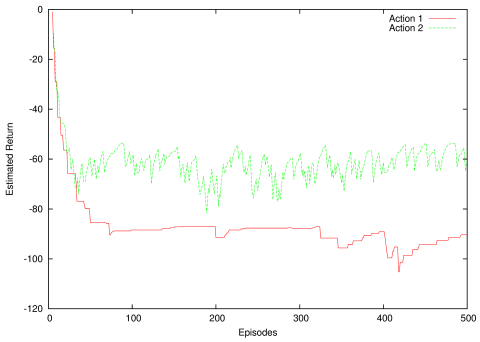



Figure 5: The estimated return with state aggregation for the most frequent initial belief state.

one of the three rooms $A$, $B$, or $C$, starting from the preceding one in a loop.

The state of the environment changed every 5 iterations of the tasks: some randomly chosen doors in {A1,A2,A3,B1,B2,B3,C1,C2} were closed and others were opened, to simulate the effect of people in the real environment. When approaching a door the robot can sense its state, and therefore update the knowledge base following the changes in the environment. The parameter were $\mu = 1.5$ for DARLING, and $\alpha = 0.8$, $\gamma = 0.9999$, $\lambda = 0.9$, and $\epsilon = 0.15$ for Sarsa.

Each simulated trial took about 14 hours in real time, which corresponds to 2.5 times as much simulated time. For this reason we could not run as many trials as with the grid-world domain. For this experiment, we show the impact of state aggregation on the estimate of the value function in the initial state, which converges to the robot's estimate for the whole task. For each task we identified the aggregated belief state which was the most frequent initial state. Let this belief state be $b_i$. Then we determined which states in $c(b_i)$ were the initial state when the agent was not performing state aggregation. The single most common aggregate belief state corresponds to 11 initial states for Task 1, 6 for Task 2, and again 11 for Task 3. It means that the agent not performing state aggregation had to learn the same values for the same actions 11, 6, and 11 times respectively just for the initial state, while it only had to learn them once while doing state aggregation. The estimates learned for state aggregation are shown in Figure 5, while the estimates without state aggregation are shown, for each of the 11 states that should have been aggregated, in Figure 4. Note that the estimates converge for state 2, the most frequent, while for the other states they are still converging. For all these other states, using the aggregated knowledge would provide a correct estimate, while the value of those actions had to be relearned.

## 6 Conclusion

We propose a method to perform state aggregation based on reasoning in answer set programming. The method allows the robot to realize, before execution, what pieces of information are certainly going to be irrelevant for learning an optimal policy. We demonstrated the approach on two domains, one of which is a realistic simulation of a service robot in an office environment. We show how much can be gained by doing state aggregation, since just in the initial state of one of the three tasks performed, the agent had to relearn the same action values 11 times. This learning effort could be spared if the robot realized that those 11 initial states are actually equivalent for the task at hand. This ability is going to be crucial for service robots which will deal with constantly increasing amounts of information.

## 7 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.

## References


[Boutilier *et al.*, 2000] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

[Chapman and Kaelbling, 1991] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *International Joint Conference on Artificial Intelligence*, volume 91, pages 726–731, 1991.

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

[Gebser *et al.*, 2011] 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.

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

[Givan *et al.*, 2003] Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1):163–223, 2003.

[Jong and Stone, 2005] Nicholas K Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *International Joint Conference on Artificial Intelligence*, pages 752–757. Citeseer, 2005.

[Li *et al.*, 2006] Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstraction for mdps. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (ISAIM-06)*, 2006.

[Lifschitz, 2008] 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.

[Seijen and Sutton, 2014] 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.

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