UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Learning relational models with human interaction for planning in robotics

David Martínez Martínez

Universitat Politècnica de Catalunya

Doctoral Programme

Automatic Control, Robotics and Computer Vision

Ph.D. Thesis

Learning Relational Models with Human Interaction for Planning in Robotics

David Martínez Martínez

Advisors:
Guillem Alenyà and Carme Torras

Barcelona, December 2016

Learning Relational Models with Human Interaction for Planning in Robotics

A thesis submitted to the Universitat Politècnica de Catalunya
to obtain the degree of Doctor of Philosophy

Doctoral programme:
Automatic Control, Robotics and Computer Vision

This thesis was completed at:
Institut de Robòtica i Informàtica Industrial, CSIC-UPC

Thesis advisors:
Guillem Alenyà and Carme Torras

LEARNING RELATIONAL MODELS
WITH HUMAN INTERACTION
FOR PLANNING IN ROBOTICS

David Martínez Martínez

Abstract

Automated planning has proven to be useful to solve problems where an agent has to maximize a reward function by executing actions. As planners have been improved to solve more expressive and difficult problems, there is an increasing interest in using planning to improve efficiency in robotic tasks. However, planners rely on a domain model, which has to be either handcrafted or learned. Although learning domain models automatically can be costly, recent approaches provide good generalization capabilities and integrate human feedback to reduce the amount of experiences required to learn.

In this thesis we propose new methods that allow an agent with no previous knowledge to solve certain problems more efficiently by using automated planning. First, we show how to apply probabilistic planning to improve robot performance in manipulation tasks (such as cleaning the dirt or clearing the tableware on a table). Planners obtain sequences of actions that get the best result in the long term, beating reactive strategies.

Second, we introduce new reinforcement learning algorithms where the agent can actively request demonstrations from a teacher to learn new actions and speed up the learning process. In particular, we propose an algorithm that allows the user to set the minimum sum of rewards to be achieved, where a better reward also implies that a larger number of demonstrations will be requested. Moreover, the learned model is analyzed to extract the unlearned or problematic parts of the model. This information allow the agent to avoid irrecoverable errors and to provide guidance to the teacher when a demonstration is requested.

Finally, a new domain model learner is introduced that, in addition to relational probabilistic action models, can also learn exogenous effects. This learner can be integrated with existing planners and reinforcement learning algorithms to solve a wide range of problems.

In summary, we improve the use of learning and automated planning to solve unknown tasks. The improvements allow an agent to obtain a larger benefit from planners, learn faster, balance the number of action executions and teacher demonstrations, avoid irrecoverable errors, interact with a teacher to solve difficult problems, and adapt to the behavior of other agents by learning their dynamics. All the proposed methods were compared with state-of-the-art approaches, and were also demonstrated in different scenarios, including challenging robotic tasks.

Keywords: learning models for planning, model-based reinforcement learning, active learning, probabilistic planning, robotics.

Resumen

La planificación automática ha probado ser de gran utilidad para resolver problemas en los que un agente tiene que ejecutar acciones para maximizar una función de recompensa. A medida que los planificadores han sido capaces de resolver problemas cada vez más complejos, ha habido un creciente interés por utilizar dichos planificadores para mejorar la eficiencia de tareas robóticas. Sin embargo, los planificadores requieren un modelo del dominio, el cual puede ser creado a mano o aprendido. Aunque aprender modelos automáticamente puede ser costoso, recientemente han aparecido métodos que permiten la interacción persona-máquina y generalizan el conocimiento para reducir la cantidad de experiencias requeridas para aprender.

En esta tesis proponemos nuevos métodos que permiten a un agente sin conocimiento previo de la tarea resolver problemas de forma más eficiente mediante el uso de planificación automática. Comenzaremos mostrando cómo aplicar planificación probabilística para mejorar la eficiencia de robots en tareas de manipulación (como limpiar suciedad o recoger una mesa). Los planificadores son capaces de obtener las secuencias de acciones que producen los mejores resultados a largo plazo, superando a las estrategias reactivas.

Por otro lado, presentamos nuevos algoritmos de aprendizaje por refuerzo en los que el agente puede solicitar demostraciones a un profesor. Dichas demostraciones permiten al agente acelerar el aprendizaje o aprender nuevas acciones. En particular, proponemos un algoritmo que permite al usuario establecer la mínima suma de recompensas que es aceptable obtener, donde una recompensa más alta implica que se requerirán más demostraciones. Además, el modelo aprendido será analizado para identificar qué partes están incompletas o son problemáticas. Esta información permitirá al agente evitar errores irrecuperables y también guiar al profesor cuando se solicite una demostración.

Finalmente, se ha introducido un nuevo método de aprendizaje para modelos de dominios que, además de obtener modelos relacionales de acciones probabilísticas, también puede aprender efectos exógenos. Mostraremos cómo integrar este método en algoritmos de aprendizaje por refuerzo para poder abordar una mayor cantidad de problemas.

En resumen, hemos mejorado el uso de técnicas de aprendizaje y planificación para resolver tareas desconocidas a priori. Estas mejoras permiten a un agente aprovechar mejor los planificadores, aprender más rápido, elegir entre reducir el número de acciones ejecutadas o el número de demostraciones solicitadas, evitar errores irrecuperables, interactuar con un profesor para resolver problemas complejos, y adaptarse al comportamiento de otros agentes aprendiendo sus dinámicas. Todos los métodos propuestos han sido comparados con trabajos del estado del arte, y han sido evaluados en distintos escenarios, incluyendo tareas robóticas.

Acknowledgements

I would like to thank my supervisors, Guillem Alenyà and Carme Torras, for their valuable guidance. Their support was essential to get to this point and I owe them this opportunity. I am also very grateful to Katsumi Inoue and Tony Ribeiro, who welcomed me to work with them at the National Institute of Informatics during several months in what became a very fruitful research stay.

Moreover, I had a great experience at both IRI and NII, and I also want to thank all the colleagues, researchers and staff that made this years a great memory. And last but not least, I would like to thank my friends and family who have supported me all these years.

This work has been partially supported by the following:

FPU12-04173 The Spanish Ministry of Education, Culture, and Sport via a FPU doctoral grant.

FP7-ICT2009-6-269959 The EU Project IntellAct.

201350E102 The CSIC project MANIPlus.

TIN2014-58178-R The MINECO project RobInstruct.

The latest version of this thesis can be obtained through
`http://www.iri.upc.edu/thesis/show/78`.

Contents

1 Introduction

Robots are capable of performing difficult actions, but they lack the cognitive abilities to solve complex tasks that require specific action sequences. There are many tasks where reactive strategies do not suffice, and designing a specific algorithm to solve each single task is not a desirable situation.

Automated planning and scheduling is a branch of AI that focuses on obtaining strategies or action sequences to solve complex tasks. Planners require a state and a model to reason about the action sequence that maximizes the reward. The state represents the agent and the environment in which the task is being executed, while the model represents how the state changes when actions are executed. The difficulty of the planning process depends on the assumptions taken about the states and the model. In this thesis, we consider probabilistic relational models with actions and exogenous effects, but we will assume that states are completely observable and discrete.

Planners rely on a model that has to be either handcrafted or learned. Manually coding models is a very tedious and error-prone task that requires specific technical knowledge about the platform being used. Therefore, it is desirable to have the option of autonomously learning models. However, learners cannot tackle yet the expressive models that planners do, which forces complex models to be mostly handcrafted. In this thesis we propose a new algorithm that can learn relational probabilistic models with action effects and exogenous effects. Such models allow robots to reason about more challenging problems:

- Uncertainty, which is ubiquitous in robotics as actions may fail or produce unexpected effects.

- Relational models that permit generalizing between different objects, which reduces significantly the experiences required to learn a model.

- Exogenous effects, which are common in real-world domains where the robot is not

isolated. Different agents, such as people or other robots, may move and interact with the environment to change the state.

In addition to a planner and a learner, we need an algorithm that obtains the set of experiences required to learn the model. Reinforcement learning (RL) is a branch of machine learning concerned with using experience gained through interacting with the world to improve a system's ability to make behavioral decisions (Littman, 2015). Model-free RL approaches are very efficient computationally but slow to learn as they require more experiences. In model-based RL, experiences are used to estimate a transition model, which is then used by planners to predict the long-term outcomes of the actions. Model-based methods are computationally intensive, but squeeze the most out of each experience. As the tasks that we are tackling have actions that are slow to execute, model-based approaches are more appropriate.

Model-based RL algorithms require a strategy to balance exploration and exploitation. Exploration is selecting actions to learn the dynamics related to unknown parts of the model, thereby allowing better policies to be obtained in the future. Exploitation consists in selecting actions to maximize rewards based on the current model. Some approaches provide good generalizations to reduce the exploration required (Lang et al., 2012), but still a lot of exploratory actions are required in complex domains to obtain good plans.

Active Learning (AL) is a special case of semi-supervised machine learning in which a learning algorithm is able to interactively query the user to request demonstrations. AL algorithms learn faster than RL ones because they do not have to explore states with low rewards (Chernova and Veloso, 2009). However, they continuously require the help of a teacher.

In this thesis, we take the different approach of combining both RL and AL. The idea is to use mostly RL to learn the model autonomously, but also to apply AL to quickly learn and overcome specific difficult problems. The result is that, by requesting a few demonstrations, a lot of exploration and low rewards can be avoided. When introducing a teacher in the RL loop, the challenge is to decide when to request a demonstration. Moreover, a more informative human-agent interaction would also help the teacher to propose better demonstrations.

Finally, the proposed algorithms are analyzed in robotic and simulated tasks. Simulated experiments can be repeated easily to get meaningful statistics, so they are the most appropriate to evaluate the performance and learning rate of an algorithm. In contrast, robotic tasks provide more challenging scenarios where the algorithms face noisy complex dynamics, but experiments involving robots are usually very time consuming to repeat and it can be difficult to repeat the initial conditions.

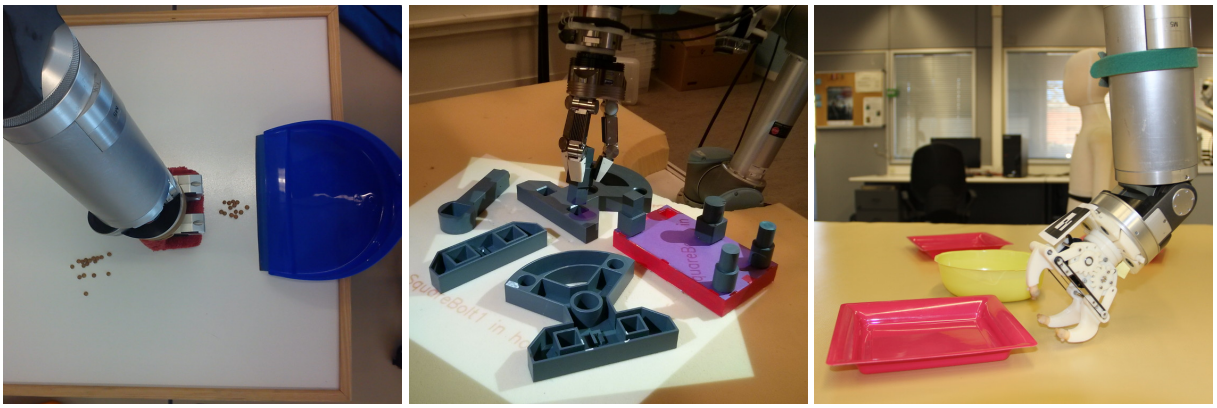


Figure 1.1: Robotic tasks. **Left:** The WAM robot is cleaning the dirt on a surface. **Center:** The UR5 robot is assembling the Cranfield benchmark. **Right:** The WAM robot is clearing the tableware on a table.

Motivation

The motivation of this PhD is to improve the performance of tasks through automated planning. However, the tasks that we tackle are not known beforehand, so we need an autonomous learner. This learner should request demonstrations from a teacher to learn new actions, and also execute such actions to improve its model. Moreover, the agent should learn from its errors, and avoid repeating them.

To evaluate the proposed algorithms, we considered two types of experiments. First, several domains from the International Probabilistic Planning Competition were used as a benchmark to compare against other state-of-the-art algorithms. However, our aim was to use those algorithms also in real domains, more precisely to robotic tasks. Therefore, we considered the following robotic tasks:

- The assembly of industrial pieces with the help of a teacher, which was part of the IntellAct EU FP7 project. The robot starts with no previous knowledge and has to learn how pieces are placed, the precedence constrains between the pieces, and how to undo wrong assemblies (Fig. 1.1-center). Furthermore, to get the most informative teacher demonstrations, the robot also provides feedback about the unlearned parts of the model so that the needed demonstrations are obtained.

- Cleaning the dirt on a surface. The robot has two types of actions: pick up actions that directly remove dirt, and grouping actions which move dirt together so that it can be picked up easily. It is not trivial to decide which actions to execute, as there is a lot of uncertainty in the actions, specially if there is a lot of dirt (Fig. 1.1-left).

- Clearing the tableware on a table. The robot piles tableware together and brings it to the kitchen. The challenge is creating the largest piles that are stable, so that the number of trips to the kitchen is minimized and the tableware does not fall. A more complex version of this problem includes external agents, such as people bringing new tableware or other robots helping to complete the task (Fig. 1.1-right).

Contributions

This thesis combines reinforcement and active learning to quickly obtain domain models for safe task execution in robotics. The probabilistic relational models learned can be used by planners to accomplish tasks in collaboration with external agents. Below we list the contributions that have been achieved:

1. We have shown that probabilistic planning can solve complex tasks faster than reactive strategies, while still providing a similar performance for easy tasks (Martínez et al., 2015a, 2013). Moreover, we proved experimentally that replanning every few actions was highly recommendable in noisy environments, and that the trade-off between planning time and plan quality should be adapted to the complexity of each problem. These results are presented in Chapter 3.

2. We have proposed the REX-D algorithm that combines RL and AL to learn tasks (Martínez et al., 2014a, 2016a). It is a RL approach that requests teacher demonstrations to quickly solve difficult problems and extend its repertoire of actions. The result is that, when compared with RL algorithms, REX-D requires much fewer action executions to learn a model that can solve the task. On the other hand, if compared with AL approaches, REX-D requests just a few demonstrations to learn, as most of the model is learned through RL. Moreover, we show that REX-D can analyze its internal model to issue more informative demonstration requests, which makes the system more accessible to non-expert teachers. REX-D is presented is Section 4.3.

3. We have extended REX-D to identify and avoid dangerous effects (Martínez et al., 2014b, 2015b). The interaction with the teacher is essential to solve the cases where safe alternatives could not be found autonomously. This method is presented is Section 4.4.

4. REX-D has also been extended into V-MIN (Martínez et al., 2015c), which is a more general algorithm where the user selects the minimum target value (sum of rewards) to be achieved. In Section 4.5 we present it and show its main advantages, that can be summarized as:

- REX-D is limited to goal-oriented tasks, but V-MIN can be applied to any reward-based task.
- The teacher can select the quality of the model to be learned (high-quality models take longer to be learned than low-quality ones).
- V-MIN can adapt easier to changes in the task while reusing its previous knowledge.

5. We have proposed a learner that obtains a relational probabilistic model with action effects and exogenous effects that explains a set of given experiences (Martínez et al., 2016b, 2015d) (Chapter 5). This work improves previous approaches that could not tackle exogenous effects (Pasula et al., 2007). We have also shown that this learner can be integrated in V-MIN to learn a wider number of domains (Martínez et al., 2017).

2

Preliminaries

In this chapter we present the background used to define the methods proposed in this thesis, and the domains in which the experiments will be performed.

Background

In this section we present the formulation that we will use throughout this thesis, as well as the related tools and techniques.

Relational Formulation

A relational representation is used to define the models, planners and learners used in this thesis. This formulation assumes complete observability and probabilistic effects.

Literals l_i are expressions of the form $(\neg)p(t_1, ..., t_m)$ where p is a predicate symbol, $(\neg)$ represents that the atom may be optionally negated, and t_i are the terms. Terms can be variables, which have a preceding "?" symbol (e.g. ?X), and can also be objects, which are represented without an "?" symbol (e.g. box1). We use a relational representation where expressions take objects as arguments to define their grounded counterparts. A state s is defined as a conjunction of grounded literals that follow the closed world assumption $s = l_1^g, ..., l_N^g$.

Experiences $E = e_1, e_2, ...$ are defined as triples $e = (s, a, s')$ where s' is the successor state of s after executing a. A successor state s' is obtained by applying all grounded operators or rules to (s, a).

Noisy Deictic Rules

A Noisy Deictic Rule (NDR) rule r is defined as

$$r(t_1, ..., t_n) : action(r) \wedge pre(r) \rightarrow \begin{cases} p(r,1) & : \textit{eff}(r,1) \\ \cdots & : \cdots \\ p(r,n_r) & : \textit{eff}(r,n_{\textit{eff}}) \\ p(r,0) & : \textit{eff}(r,0), \end{cases} \tag{2.1}$$

where t_i are the terms, $action(r)$ is the action that the rule represents, $pre(r) = l_1 \wedge \cdots \wedge l_m$ are a set of literals that represent the preconditions for the rule to be applicable, $\textit{eff}(r,i)$ are the effects that define the set of literals that are changed in the state with probability $p(r,i)$ when the rule is applied, and $\textit{eff}(r,0)$ is the noisy effect that represents all other, unmodeled, rare and complex effects. The sum of effect probabilities must be $\sum_i p(r,i) = 1$.

A NDR rule represents only one action. However, each action may be represented by several rules, where each has different preconditions. All the rules defining one action have disjoint preconditions, and therefore, each state-action pair (s,a) is covered by just one rule r. This is required because planners can only work with conflict-free rules. A planner has to know precisely the expected effects of applying a rule. If two different rules were to make conflicting changes, the effects would be undefined.

Planning Operators

A planning operator $o \in \mathcal{O}$ defines how a literal changes based on a set of preconditions. Operators take the form

$$o(t_1, ..., t_n) = l_h : p(o) \leftarrow l_1 \wedge \cdots \wedge l_m, (a) \tag{2.2}$$

where l_h is the head of the operator, $p(o)$ is the probability of l_h being in the next state given that the body and the action are satisfied, $l_1 \wedge \cdots \wedge l_m$ are the literals in the body, (a) is an optional action, and t_i are the terms that may appear in the head, body and action. The action is optional so that operators can capture both action effects when there is an action, and exogenous effects when there is no action. Note that operators are not Horn clauses as negation can appear in both the body and the head.

We require operators to be mutually exclusive, there cannot be two operators with the same head atom that cover the same (s,a) as their heads may conflict. This is required because planners can only work with conflict-free operators. One example of such conflict would be $[o_1(obj1) = at(obj1) : 0.8 \leftarrow ...]$ and $[o_2(obj1) = \neg at(obj1) : 0.6 \leftarrow ...]$ where both heads cannot

hold at the same time as one contradicts the other.

To define the transition model we will use either NDR rules or planning operators. The main differences between them are:

- NDR rules have several effects, each with several literals. Therefore, they can represent literals that change together. Planning operators only have a single effect with one literal.
- NDR rules must represent an action, and only one NDR rule may be applied at each time step. Planning operators can optionally represent an action, and different operators may be applied at the same time step. Therefore, planning operators can represent exogenous effects while NDR rules cannot.

As a result, we will use planning operators in tasks with exogenous effects, and NDR rules will be the preferred choice for tasks without exogenous effects.

Grounding and Applying Operators and Rules

A grounded operator only has objects as terms. If an operator o has n variables, its groundings $Gr(o)$ are a set of operators, each taking one of the possible combinations of n objects.

Example 2.1. Having the objects $\{a, b, c\}$ and the operator $o1(?X, ?Y) = at(?Y) : 0.8 \leftarrow road(?X, ?Y) \wedge at(?X)$, the possible groundings can be obtained by substituting $?X$ and $?Y$ for every permutation of 2 objects. One grounding would be: $o1(a, c) = at(c) : 0.8 \leftarrow road(a, c) \wedge at(a)$.

The transition dynamics are defined by a set of planning operators $\mathcal{O}$. A grounded operator o_g is said to cover a state-action pair (s, a) when the literals of the body are in s, and the optional action of the operator is either a, or the operator has no action: $cov(o_g, s, a) = (body(o_g) \subset s) \wedge ((action(o_g) = a) \vee (action(o_g) = \varnothing))$.

Likewise, a non-grounded operator o covers (s, a) if one of its groundings does: $cov(o, s, a) = \exists\, o_g \in Gr(o) \mid cov(o_g, s, a)$.

The same operations that we have shown for planning operators can be applied to NDR rule. The only difference would be that a grounded rule r_g covers a state-action pair (s, a) when the literals in the preconditions are in s, and the action a is equal to the rule's action: $cov(r_g, s, a) = (pre(r_g) \subset s) \wedge (action(r_g) = a)$.

Calculating the Likelihood

A successor state s' is obtained by applying all groundings of all operators to (s, a). When a grounded operator o_g is applied to (s, a), its head is added to the state s' with a probability $p(o_g)$ if $cov(o_g, s, a)$.

If s' is a successor state of s, we define $changes(s, s')$ as the set of literals $\{c \in s', c \notin s\}$. Given an experience $e = (s, a, s')$ and a grounded operator o_g, a change $c \in changes(s, s')$ has a likelihood

$$P(c \mid o_g) = \begin{cases} p(o_g), & cov(o_g, s, a) \wedge (c = head(o_g)) \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

A set of non-grounded operators $\mathcal{O}$ gives the following likelihood to a change c:

$$P(c|\mathcal{O}) = \begin{cases} P(c|o_g), & \exists!\, o_g \in Gr(\mathcal{O}) \mid P(c|o_g) > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{2.4}$$

where $\exists!$ is the operator for uniqueness quantification. If more than one operator covers the same change given the same state-action pair, there is a conflict and the behavior is undefined, so a likelihood of 0 is given.

Finally, a set of planning operators $\mathcal{O}$ covers an experience $e = (s, a, s')$ with a likelihood

$$P(e \mid \mathcal{O}) = \prod_{c \in \text{changes}(e)} P(c \mid \mathcal{O}, s, a). \tag{2.5}$$

Likewise, a grounded NDR rule r_g covers an experience $e = (s, a, s')$ with a likelihood

$$P(e \mid r_g) = \sum_i \begin{cases} p(r_g, i), & cov(r_g, s, a) \wedge (s' \supset \mathit{eff}(r_g, i)) \\ 0, & \text{otherwise,} \end{cases} \tag{2.6}$$

and a set of NDR rules R has a likelihood

$$P(e|R) = \begin{cases} P(e|r_g), & \exists!\, r_g \in Gr(R) \mid P(e|o_g) > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

Propositional Formulation

On the propositional level, multi-valued atoms m_i are expressions of the form $p = x$, where p is a predicate, x is the value, and atoms have no terms. A state is a conjunction of propositional multi-valued atoms $s = m_1, m_2, ..., m_n$ where every predicate must appear only once (i.e. $\forall\, m_i \in s,\ \nexists\, x, y \mid ((m_i = x) \wedge (m_i = y)),\ x \neq y$).

The propositional formulation will be used only in Chapter 5 with LFIT algorithm (Inoue et al., 2014).

Markov Decision Processes

Fully-observable planning problems with uncertainty can be described formally with Markov Decision Processes (MDP). A finite MDP is a five-tuple $\langle S, A, T, \mathcal{R}, \gamma \rangle$ where:

- S is a set of discrete states.

- A is a set of actions that an agent can perform.

- $T : S \times A \times S \rightarrow [0, 1]$ is the transition model that describes the probability of obtaining a successor state by executing an action from a given state. Note that in this thesis we will use a set of planning operators $\mathcal{O}$ or a set of NDR rules R to define the transition model.

- $\mathcal{R} : S \times A \rightarrow \mathbb{R}$ is the reward function.

- $\gamma \in [0, 1)$ is a discount factor that measures the degradation of future rewards.

A policy π is a function $\pi : S \rightarrow A$ that specifies which action to execute when in state s. The value function $V^{\pi}(s) = \mathrm{E}[\sum_t \gamma^t \mathcal{R}(s_t) \mid s_0 = s, \pi]$ is the sum of expected rewards when applying the policy π from state s. The goal is to find a policy π that maximizes the value function $V^{\pi}(s)$.

Model Learning

Machine learning techniques can be used to learn models for planning from a set of experiences. We have used Pasula et al. (2007)'s learning algorithm in some of our algorithms as it was the best approach for probabilistic relational domains. Eventually, we proposed a new learning algorithm (Chap. 5) that can learn more expressive models.

Pasula et al. (2007) use a greedy heuristic search to obtain a set of planning operators $\mathcal{O}$ that minimize a score function. The algorithm starts with an arbitrary set of candidates, and iteratively modifies this set and checks if the score function improves. This process is repeated until the set cannot be further improved. The algorithm is very fast but has local minimums.

The score function for a set of experiences E is

$$s(\mathcal{O}, E) = \mathop{\mathrm{E}}_{e \in E}[\log(P(e|\mathcal{O}))] - \alpha \frac{Pen(\mathcal{O})}{Conf(E, \epsilon)}, \tag{2.8}$$

where $\alpha > 0$ is a scaling parameter, and the penalty term $Pen(\mathcal{O}) = \sum_{o \in \mathcal{O}} |body(o)|$ is the number of atoms in the operator bodies. With this score function, the algorithm looks for sets of operators that maximize the likelihood and minimize the number of atoms in the operators.

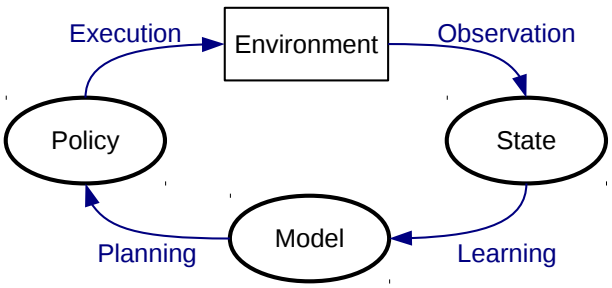


Figure 2.1: Overview of a model-based RL algorithm.

Reinforcement Learning

We use model-based RL to tackle unknown problems. In this thesis we will consider the case where the reward function $\mathcal{R}$ is given and the transition model T is unknown. Figure 2.1 shows an overview of a standard model-based RL algorithm:

1. Observations of the environment are used to update the state.
2. Previous experiences consisting on triples $e = (s, a, s')$ are used to learn a model.
3. The planner obtains a policy with the model.
4. The action selected by the policy is executed.

The goal of the agent is to maximize $V(s)$, and it can employ two strategies to accomplish it:

- Exploration: Execute actions to explore unvisited parts of the model. The more complete a model is, the better the policies obtained by the planner, which improves long-term rewards.
- Exploitation: Execute actions to maximize the sum of discounted rewards based on the current model.

In RL, a very important problem is the exploration-exploitation dilemma, which involves finding a balance of sufficient exploration to obtain a good model without consuming too much time addressing the low-value parts of the state.

Active Learning

In active learning, an agent can request demonstrations from the teacher to learn. In our work, when a demonstration is requested, the teacher selects an action from the optimal policy $\pi^*(s)$, and tells the agent the name of the action, its parameters and how to perform it.

In robotics, standard human-robot interaction capabilities such as speech and visual information provide the communication between the teacher and the robot. Moreover, learning from demonstration (Argall et al., 2009) can be used to learn new actions.

Planning Excuses

Whenever the system fails to plan, information can be extracted about the failure using excuses (Göbelbecker et al., 2010; Menezes et al., 2012). Excuses are changes to the initial state that would make the task solvable, and thus they indicate the important literals that cause the planner to fail.

Definition 2.1. (Excuse) Given an unsolvable planning task, using a set of objects C_{s_0} and an initial state s_0, an excuse is a pair $\varphi = \langle C_\varphi, s_\varphi \rangle$, which makes the task solvable, where C_φ is a new set of objects and s_φ is a new initial state.

Excuses can be classified as acceptable, good, or perfect, as follows.

- Acceptable excuses change the minimum number of literals in the initial state. An excuse φ is acceptable iff $\forall \varphi', C_\varphi \subseteq C_{\varphi'}$ and $s_0 \triangle s_\varphi \subseteq s_0 \triangle s_{\varphi'}$ (where $\triangle$ denotes the symmetric set difference).

- Good excuses are acceptable excuses with changes that cannot be explained by another acceptable excuse.

- A perfect excuse is a good excuse that obtained the minimal cost using a cost function.

Applying goal regression over all acceptable excuses would be highly suboptimal, so a set of good excuse candidates are generated and checked with the planner (Göbelbecker et al., 2010).

Candidates for good excuses can be obtained under certain assumptions using a causal graph and a domain transition graph (Helmert, 2006), which are generated with the planning operator set $\mathcal{O}$. A causal graph $CG_\mathcal{O}$ is a directed graph that represents the dependencies of literals between each other. A domain transition graph G_l of a literal l is a labeled directed graph that represents the possible ways that the groundings of the literal can change and the conditions required for those changes.

Definition 2.2. A causal graph $CG_\mathcal{O}$ is a directed graph that represents the dependencies of literals between each other. An arc (u, v) exists when $u \in body(o)$ and $v \in head(o)$ for an operator $o \in \mathcal{O}$.

Definition 2.3. A domain transition graph G_l of a literal l is a labeled directed graph that represents the possible ways that the groundings of the literal can change and the conditions

required for those changes. An arc (u, v) exists when there is a grounded operator $o_g \in Gr(\mathcal{O})$ such that u and v are groundings of l, $u \in body(o_g)$ and $v \in head(o_g)$. The label comprises the literals $body(o_g) \setminus \{u\}$.

To restrict the number of candidates, we only consider those that are relevant to achieving the goal. Using the causal graph and the domain transition graph, the candidates can be obtained with a fix point iteration (Göbelbecker et al., 2010) by adding the literals that contribute to the goal and those that are potentially required to reach other literals added previously to the candidate set. From the set of excuse candidates, we select those that are not reachable by G_l from any literal in the current state, and those that are involved in a cyclic dependency in $CG_{\mathcal{O}}$.

Finally, the planner is used to test which of the excuse candidates should be added to obtain the best results. The best are selected as the excuses that explain the failure.

Note that operators in stochastic domains have effects with different probabilities. To generate the excuses in these conditions, low probability effects are ignored when generating the causal graph and domain transition graph.

Domains Used for Experimentation

In this section we describe the domains used to validate experimentally the proposed algorithms. There are two types of domains:

- *Domains from the International Probabilistic Planning Competition (IPPC)*. These domains are widely used in the planning community, and thus they are the best choice to compare against other state-of-the-art algorithms.

- *Robotic tasks.* They provide more challenging scenarios to challenge the effectiveness of planners and learners.

Simulated problems can be easily repeated to get meaningful statistics, so they were the appropriate tool to evaluate the performance of algorithms. In addition to IPPC domains, some robotic domains could also be executed in simulation. To automate experiments that involve interacting with a teacher, we developed an automated teacher that plans the optimal action based on a ground truth model. If more than one action was optimal, it considers the guidance provided by the learner (see Sec. 4.3.2) to choose which action to demonstrate. This automated teacher performs similarly to human teachers because it uses optimal policies.

In contrast, robotic tasks provide more realistic challenges that involve a lot of uncertainty and unforeseen situations. However, they cannot be repeated automatically, so it is more time consuming to repeat them.

Domains from the IPPC

The International Probabilistic Planning Competition (IPPC) (Vallati et al., 2015) provides a set of domains that are designed for investigating the applicability of planning techniques to a range of real-world applications.

Traditionally, PPDDL (Younes and Littman, 2004) was the standard language used for planning applications. Among other features, actions had parameters (variables that may be instantiated with objects), preconditions and effects. The effects of actions could be also conditional. At the IPPC 2011 (Coles et al., 2012) the language used in the uncertain track was changed to RDDL (Sanner, 2010), which allowed modeling a variety of new problems with stochasticity, concurrency, and a complex reward and transition structure.

NDR rules are easy to translate to PPDDL ones as both are a preconditions-effects representation, but they are difficult to translate to RDDL because every literal dynamics have to be specified separately. In contrast, planning operators are easy to transform to a RDDL representation, but exogenous effects cannot be represented directly in PPDDL (a rule for each combination of action effects and exogenous effects would have to be created).

In this thesis we will use domains from the IPPC 2008 as they have a wider support from planners and learners, and also from the IPPC 2014 in cases where the learners support them. Most IPPC 2014 domains include exogenous effects that require a RDDL representation and a learner that supports them.

Triangle Tireworld

In this domain, a car has to move to its destination, but it has a probability of getting a flat tire while it moves. The car starts with no spare tires but it can pick them up in some locations. The actions available in this domain are: a “Move” action to go to an adjacent position, a “Change Tire” action to replace a flat tire with a spare tire, and a “Load Tire” action to load a spare tire into the car if there are any in the current location. The main difficulty in the *Triangle Tireworld* domain is the dead end when the agent gets a flat tire and no spare tires are available. Safe and long paths exist with spare tires, but the shortest paths do not have any spare tires. Figure 2.2 shows the easiest problem of the *Triangle Tireworld* domain (problem 1 in IPPC 2008 and 2014).

This domain was present in both IPPC 2008 and 2014. The difference is that in the IPPC 2008 the problem finishes when the goal is reached, while in the IPPC 2014 there is one exogenous effect (when the goal reward is received the “goal-reward-received” literal becomes true, and the reward is no longer obtained). This subtle difference has no impact for planners, but this extra exogenous effect has to be learned by learners.

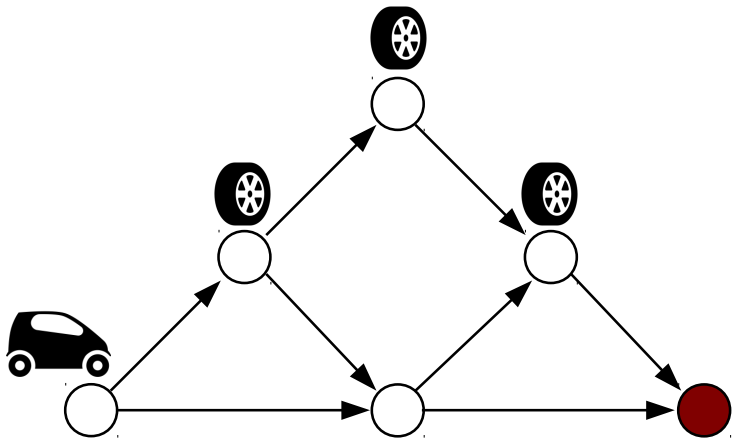


Figure 2.2: The *Triangle Tireworld* domain. The car represents the position where the robot starts, the red circle represents the goal, and the tires represent that the position contains a spare tire.

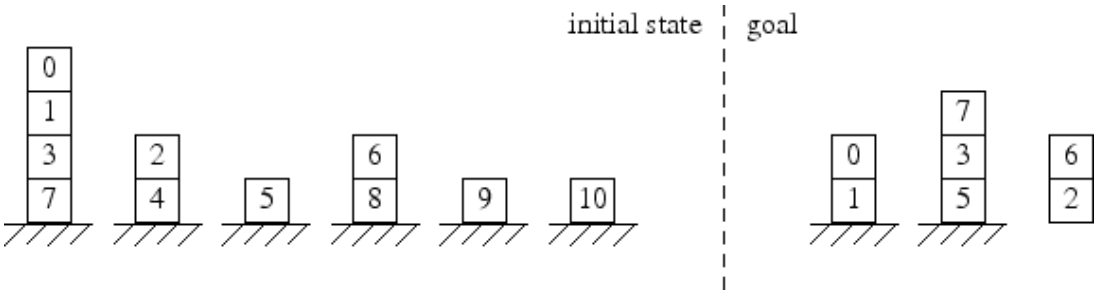


Figure 2.3: The *Exploding Blocksworld* domain. An example of an initial and a goal states is shown. The image is reproduced from (Younes et al., 2005).

Exploding Blocksworld

This domain, which was part of the IPPC 2008, is an extension of the well-known *Blocksworld* domain that includes dead ends. The robot employs "pick up" and "put on" actions to position a set of blocks in a given layout. A block that is placed on the table, or another block, has a probability of exploding and destroying the object beneath. After a block is destroyed, it cannot be picked up and no other blocks can be placed on top of it. Destroying the table also implies that the system cannot place blocks on it anymore. To solve these problems, the planner has to take care to avoid destroying the blocks that are important for reaching the solution. Figure 2.3 shows an example of a problem in the *Exploding Blocksworld* domain.

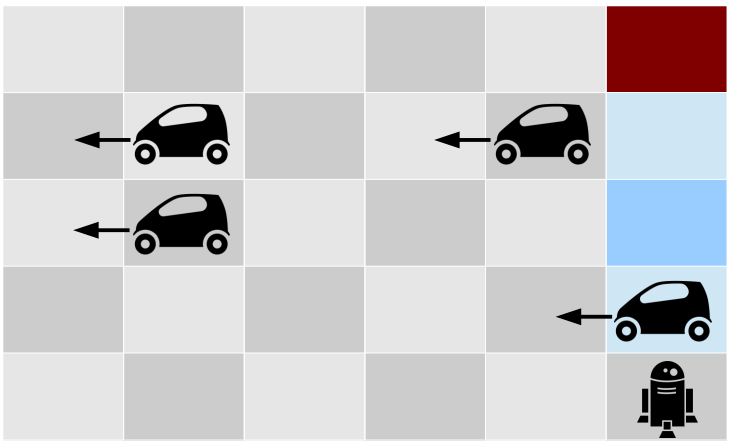


Figure 2.4: The *Crossing Traffic* domain. The robot has to reach the goal on the top right (red). However, cars appear randomly from the right side (blue) and move to the left.

Crossing Traffic

The *Crossing Traffic* domain (IPPC 2014) is a grid where a robot must get to a goal and avoid cars arriving randomly and moving left. The goal is located on the top right of the grid, and the robot starts on the bottom right position. If a car overlaps with the robot, the robot disappears and can no longer move around. The robot can "duck" underneath a car by deliberately moving right when a car is to the right of it. The robot receives -1 for every time step it has not reached the goal. The best strategy is to move first to the left to be able to see if a car is coming, and crossing whenever there are no cars in the way. Figure 2.4 shows an example of the *Crossing Traffic* domains.

Elevators

The *Elevators* (IPPC 2014) domain has a number of elevators delivering passengers to either the top or the bottom floor (the only allowable destinations). Potential passengers arrive at a floor based on Bernoulli draws with a potentially different arrival probability for each floor.

An elevator can move in its current direction if the doors are closed, can remain stationary (noop), or can open its door while indicating the direction that it will go in next (this allows potential passengers to determine whether to board or not). Note that the elevator can only change direction by opening its door while indicating the opposite direction. Figure 2.5 represents the *Elevators* domain.

A passable plan in this domain is to pick up a passenger every time they appear and take them to their destination. A better plan includes having the elevator "hover" near floors where passengers are likely to arrive and coordinating multiple elevators for up and down passengers.

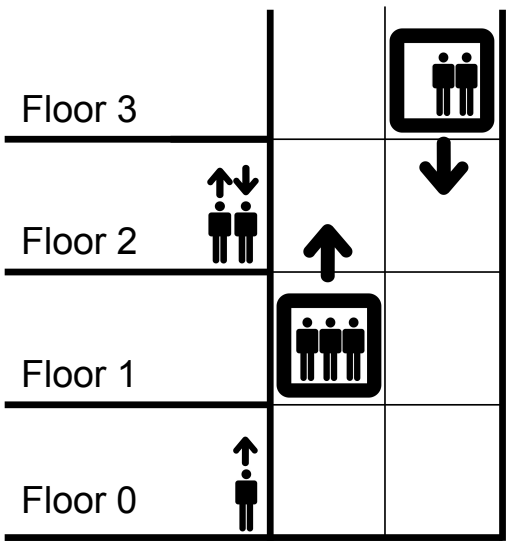


Figure 2.5: The *Elevators* domain. On the left, people are waiting for an elevator, and the arrow on each person indicates whether that person is going up or down. On the right, the elevators are carrying passengers to another floor. When an elevator stops on a floor, it has to open the doors to let people enter.

Robotic Applications

Here we describe the different robotic applications that we wanted to solve. In following chapters we will show how can they be solved with the proposed algorithms.

Cleaning Dirty Surfaces

This task consists in cleaning the dirt on a surface by using a robot arm grasping a cloth, and an optional secondary robot arm holding a dustpan. The objective of the robot is to minimize the time required to clean the surface.

The robot has a set of actions designed to clean all distributions of dirt, including dirt *grouping* actions, and dirt *cleaning* actions. Grouping actions rearrange the dirty areas on the surface, so that they become easier to clean by means of future actions, while cleaning actions directly remove the dirt. The difficulty is finding the best combination of these actions to clean the dirt. This task is explained with more detail in Sec. 3, and Fig. 2.6 shows the WAM robot cleaning with different cloth grasps.

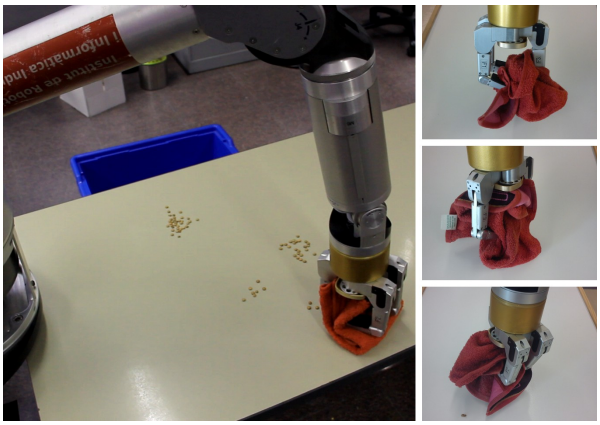


Figure 2.6: **Left:** the WAM robot arm is cleaning lentils from a table to a container. **Right:** three different grasps of the textile used for cleaning, each one leading to different behaviours of the cleaning actions.

The Cranfield Benchmark

The Cranfield benchmark involves assembling an industrial item, the parts of which are shown in Fig. 2.7-left. There are precedence constraints in the assembly that restrict the order in which the parts have to be assembled. Square pegs have to be placed before the separator, the shaft before the pendulum, and all other pieces before the front faceplate.

Figure 2.7-right shows an example of a state. The state defines which holes of the back faceplate are free and the status of each piece (i.e. if it is graspable, if it is already assembled, and if it is a peg in a horizontal position that is harder to grasp).

There is a different action to place each type of object, and the reward is obtained after completing the assembly. Moreover, some variants of this scenario are used in more difficult and interesting problems, as follows.

- Pegs are difficult to place when they are in a horizontal position. Therefore, actions are required to place them in a vertical position.

- Wrong initial states can be used, e.g., an initial state where the separator is placed before the pegs. In this case, the robot first has to remove the separator in order to place the square pegs.

- Changing some parts for new ones, such as replacing the front faceplate with another that is not compatible with a pendulum. This is an interesting problem for learning when part of the domain is already known.

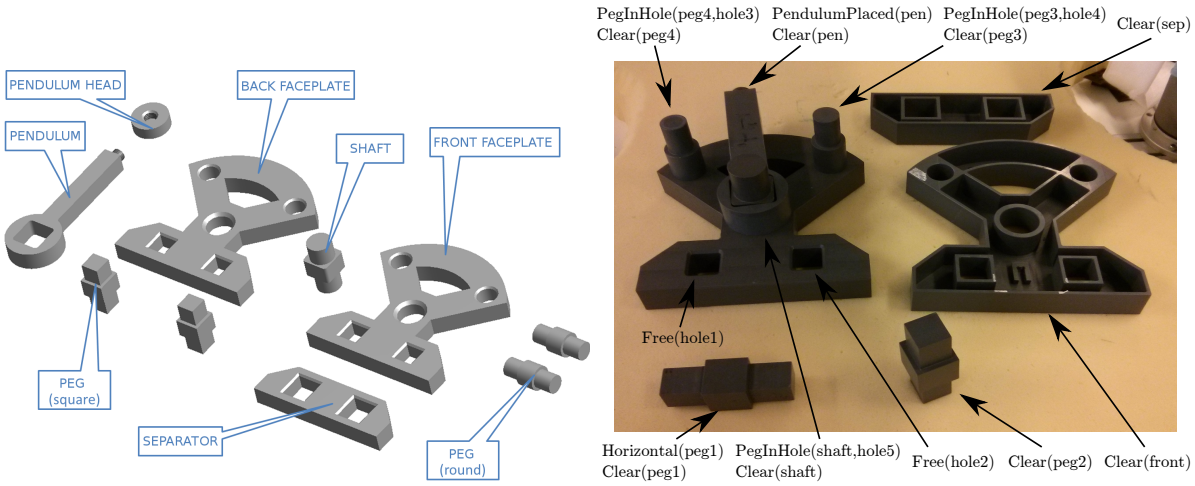


Figure 2.7: Cranfield benchmark assembly. **Left:** The different pieces used to assemble the structure. **Right:** An example of the state literals used to describe the Cranfield benchmark.

Clearing Tableware on a Table

This task represents a robotized restaurant. The robots have to take plates, cups and spoons from a table to the kitchen. However, moving to the kitchen is a costly action, and therefore the robot has to stack the tableware before taking them, minimizing the time spent.

The actions can take the top object on top of a pile and place it in another location, or move entire piles. Piles may become unstable if objects are not piled properly. For example, if a plate is placed on top of a pile containing a cup and a fork, there is a high probability that it will become unstable. To obtain stable piles, in general, plates should be placed on the bottom, cups on top of them, and cutlery on the top. Unstable piles are harder to move, and objects may fall and break. Finally, the robot has to take the whole piles to kitchen. The difficulty of this task is creating the largest possible stable piles so that the number of trips to the kitchen is minimized and no objects are broken.

The robotic setup (Fig. 2.8) consists of a robot arm equipped with a gripper, and a RGB-D Kinect camera that is positioned on the ceiling. To generate symbolic state representations for the decision maker, the perception system recognizes the tableware on the table and their relative positions, and maintains a believe state that is needed to tackle the occlusions when an object is placed on top of another. The movement primitives to execute pick and place actions are also available in the robot.

There are two variants of this task:

- An easy one where there is only one robot that has to pile and take tableware to the kitchen.

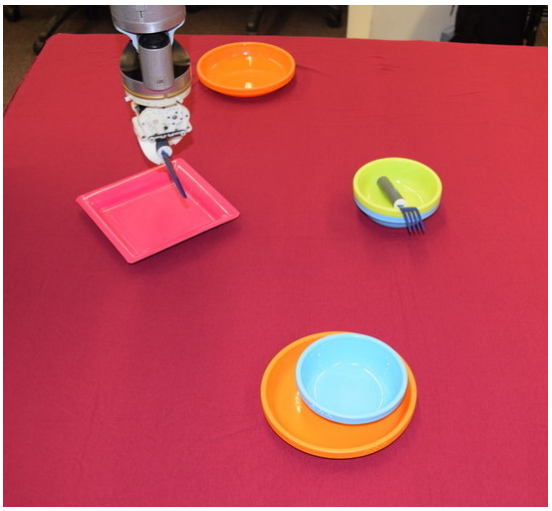


Figure 2.8: An arm manipulator is stacking the tableware to clear the table.

- A difficult one with external agents. There are mobile robots that take piles from the tables to the kitchen, and people that bring continuously new tableware to the table. We control a manipulator robot that has to make piles for the mobile robots and ensure that there is enough space for people to bring new tableware.

3
Planning in Robotics

This chapter presents a new approach to plan high-level manipulation actions for cleaning surfaces in household environments, like removing dirt from a table using a rag. Dragging actions can change the distribution of dirt in an unpredictable manner, and thus the planning becomes challenging. We propose to define the problem using explicitly uncertain actions, and then plan the most effective sequence of actions in terms of time. However, some issues have to be tackled to plan efficiently with stochastic actions. States become hard to predict after executing a few actions, so replanning every few actions with newer perceptions gives the best results, and the trade-off between planning time and plan quality is also important. Finally a learner is integrated to provide adaptation to changes, such as different rag grasps, robots, or cleaning surfaces. We demonstrate experimentally, using two different robot platforms, that planning is more advantageous than simple reactive strategies for accomplishing complex tasks, while still providing a similar performance for easy tasks. This work has been published in (Martínez et al., 2013, 2015a).

Introduction

Robots in household environments tend to move slowly to produce human safe actions, and therefore a lot of the time spent in a given task is devoted to carefully move the robot. The challenge is to produce sequences of actions that minimize the number of robot motions for a given task, and thus the overall time.

An automated planner can be used to select the best sequence of actions, the *plan*, in terms of some *metrics*. In this work we use the criterion of minimum time for the whole task, including the computing time and the execution time. Given a *state* (a representation of the environment), and a set of *rules* (definitions of the set of actions that can be executed), the planner computes the best plan to complete the task.

The robot interacts with a real environment by executing actions, and such interaction may change it in an unpredictable manner. Hence, it is desirable that the robot keeps replanning and adapts the plan to the unexpected changes that may arise.

We present a new approach to clean surfaces with plans of robot dragging actions. We have developed a system that continuously updates a state representing the dirt on a planar surface, and has a set of actions designed to clean dirt, including dirt *grouping* actions, and dirt *cleaning* actions. We show that although planners are usually applied to simulated scenarios where problems have stricter constraints (Vallati et al., 2015) and successor states can be easily obtained (Lipovetzky et al., 2015), some of them also provide good results in handling the uncertainty present in real problems (Lang and Toussaint, 2010), as the one addressed in this work. The experiments are performed in two platforms: a commercial REEM service robot, and a WAM manipulator cell, both equipped with a RGB-D camera (see Fig. 3.1), and the task is to clean lentils from a table.

Actions maintaining contact are usually implemented using force control, but force feedback is not available in the REEM robot. Here we use depth information from RGB-D cameras and deformable tools like cloth, which provides some compliance. As can be seen in Fig. 3.1 the cloth adapts to the surface, and thus, the robot can successfully move dirt with the limited resolution of RGB-D cameras. Although this alternative adds some extra uncertainty, it can be tackled with probabilistic planning without worsening much the results.

This chapter is structured as follows. Section 3.2 presents some previous work. The proposed algorithm is introduced in Sec. 3.3, where perceptions, the set of actions, and the planning strategy are presented. Section 3.4 explains the details of using a planner with uncertain actions. Section 3.5 shows the experiments in the two robot platforms, and presents a comparison between our approach and a very effective reactive strategy. Finally, Sec. 3.6 is devoted to draw some conclusions and future work.

Previous work

Previous work has already tackled the problem of surface cleaning (Kormushev et al., 2011; Sato et al., 2011), where robot skills to clean a whiteboard are presented. The robot is trained using imitation learning with hybrid position/force control to learn and execute trajectories while maintaining the force of the hand against the whiteboard. Gams et al. (2010) have also used force feedback to learn dynamic motion primitives that ensure that the robot maintains contact and applies the desired force in tasks such as wiping a table. Moreover, Nemec and Ude (2012) have proposed methodologies for sequencing motion primitives, allowing to perform more complex actions. However, in all these works, the surface cleaning strategy is fixed and

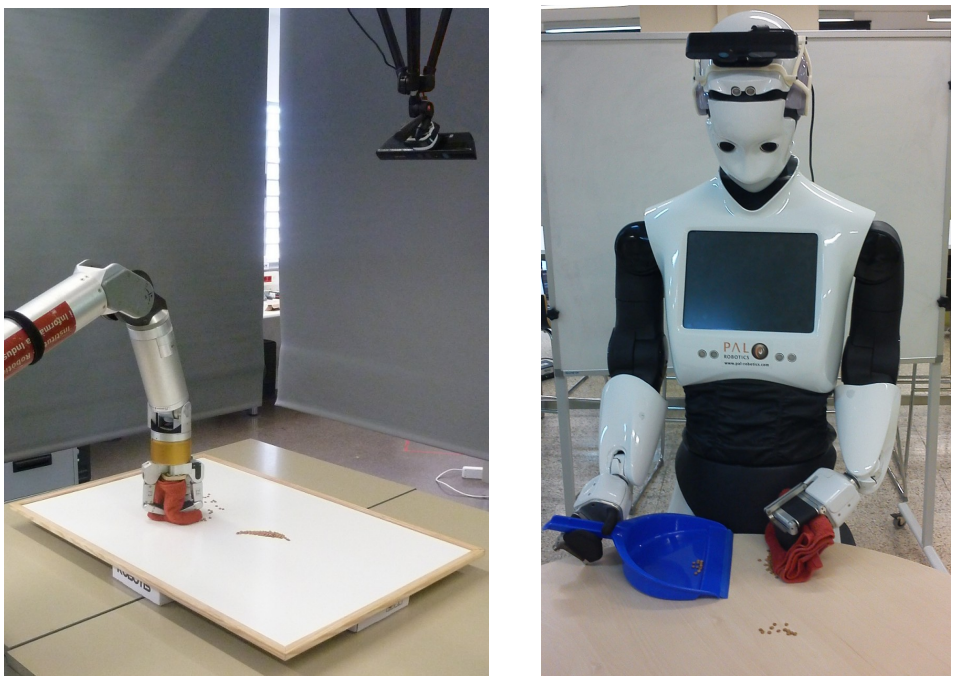


Figure 3.1: The two robotic platforms used in the experiments performing cleaning tasks. **Left:** the WAM arm manipulator. **Right:** the REEM service robot.

thus the robot is unable to adapt to different distributions of dirt, some of which could be cleaned more efficiently with simpler trajectories.

For this, a perception system is necessary to analyze the scene and select actions accordingly. Bormann et al. (2013) developed an autonomous dirt detection system, but they do not tackle the problem of cleaning. Hess et al. (2011) formulate a table cleaning problem as a Markov Decision Process (MDP). The table surface is divided into cells, and a robot displaces a vacuum cleaner to the dirty cells. Interestingly, the initial perception can mark background textured points as dirty, and the algorithm learns to separate dirt from background, as it assumes that the vacuum cleaner removes all the dirt from a visited cell. The discount factor in the MDP is set to 0, converting planning into a 1-step lookahead problem. Hess et al. (2013) provide a more complete approach. They use a grid with Poisson processes to estimate the dirt and apply a TSP solver to generate the path used to clean the surface. These approaches work well when actions are accurate and grouping actions, that change the arrangement of dirt, are not considered.

Planning with actions is challenging. Cambon et al. (2009) propose to link the symbolic description of a task with the geometric effects of the manipulation. They develop the aSyMov planner, and show the benefits of using actions to place the robot in better positions for object manipulation and to modify the environment by displacing objects. Unfortunately, aSyMov does not consider uncertainty issues, which are important in realistic environments.

Planning has also been used in manipulation tasks with uncertain perceptions and stochastic actions (Kaelbling and Lozano-Pérez, 2012). Although stochastic actions are considered, planning takes into account only the most likely effect for every action, and re-planning is triggered whenever that effect fails to occur. In contrast, we want to take into account all possible effects when planning, as they are needed to obtain optimal plans.

In probabilistic planning, one option is to *determinize* the problem and solve it using a deterministic planner, like FF-Replan (Yoon et al., 2007), that uses FF (Hoffmann and Nebel, 2001) as such deterministic planner. Since probabilities are ignored when *determinizing*, it is accepted that this option is not appropriate for domains in which the probabilities are important to get good plans (Little and Thiebaux, 2007a). In our problem, cleaning actions may lead to diverse results, and although one action may be the best for a given state, a different one may have a better overall performance in the long term.

MDPs are used to solve probabilistic planning, for which both exact and approximate solvers are available. When tackling large state spaces, two common techniques are UCT (Kocsis and Szepesvári, 2006), which finds near-optimal solutions in finite-horizon or discounted MDPs; and LRTDP (Bonet and Geffner, 2003), an optimal heuristic search algorithm.

Lang and Toussaint (2010) propose a probabilistic planner that handles uncertainty by converting the rules into a dynamic Bayesian network for state representation, and predicts the effects of action sequences by using an approximate inference method. PRADA is an online procedure that is able to get a plan for every state, and has a better performance than classic UCT and LRTDP. Although newer planners outperform it (Kolobov et al., 2012b; Keller and Eyerich, 2012), it can cope with unmodeled noise, which is usually present in robot actions, making it an appropriate planner for our task.

Proposed approach

The method proposed is aimed at cleaning a surface using a calibrated RGB-D camera, a robot arm grasping a cloth, and an optional secondary robot arm holding a dustpan.

A symbolic state s that represents the scene is obtained from the RGB-D camera, which provides depth and color data.

The robot has a set of actions consisting of sequences of movements to clean or displace dirt. Every action is represented by at least one rule, which encodes the expected effects when a set of preconditions holds.

Given a state and a set of rules defining the actions, the planner chooses a sequence of actions to clean the surface efficiently, which is then executed by the robot. Once the robot has finished, it will plan again with an updated state and execute new actions until all dirt is cleaned.

The actions generated by the planner have to be converted into motions. Depending on the action, a different type of movement will be created based on the dirty areas that it acts upon. The depth information from the camera is used to obtain the 3D positions corresponding to the dirty areas, which are then used to generate the 3D points where the robot will move to.

Perception

RGB-D cameras obtain observations containing both depth and color information. Depth is used to segment the surface to be cleaned, while color permits segmenting the dirty areas on the surface. Finally, to simplify the state, those dirty areas are represented by ellipses. The perception system is designed to be effective with different types of dirt, such as small particles like lentils or ink on a board. Figure 3.2 shows an example of the perception process, which performs the following steps:

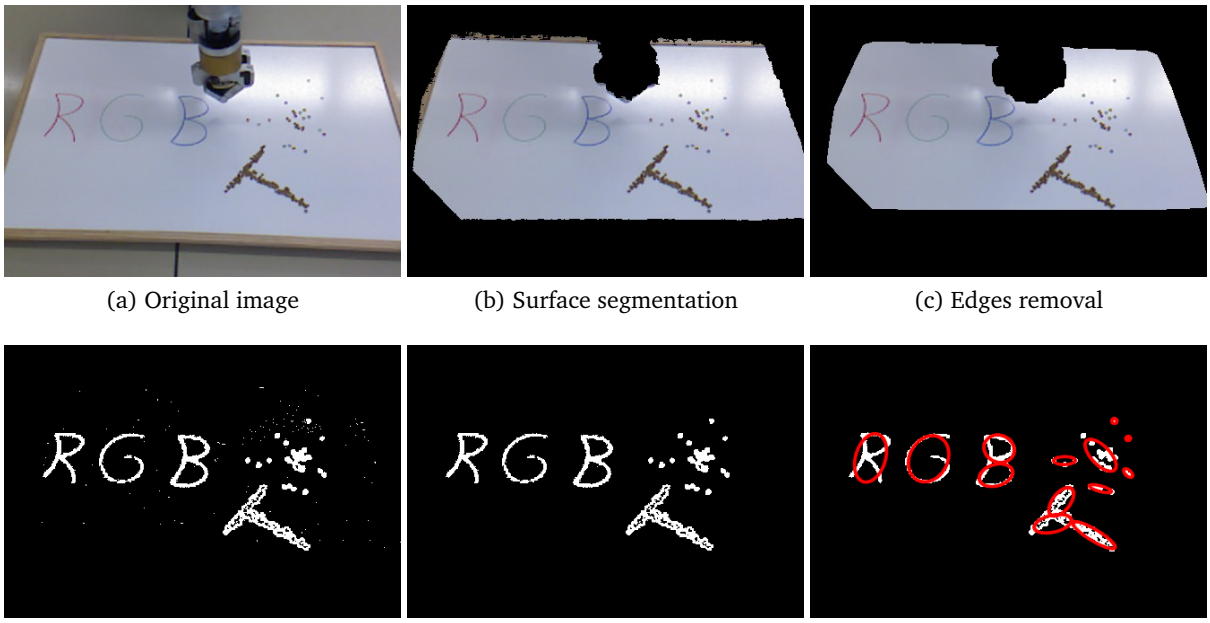
- *Surface segmentation*: Depth information is used to segment the dirty surface, which may be located in any position and orientation. For simplicity the surface is assumed to be planar, allowing us to detect it by using the RANSAC algorithm. Once detected, all points lying outside the plane will be removed (see Fig. 3.2b).

- *Edges removal*: RGB-D cameras are not accurate in the color and depth registration. This is crucial in the edges of a 3D object, where the color given to a point may be the color of a different object in its neighborhood and detected as dirt (see Fig. 3.2c). Edges are therefore removed.

- *Background subtraction*: The image is divided in different areas through region growing using relative gradients. Areas bigger than a certain threshold are considered to be background and thus removed. As a result, the remaining areas will be the dirty ones (see Fig. 3.2d).

- *Noise removal*: A median filter is applied to remove spurious points produced by the background segmentation. Moreover, a belief state is maintained for every pixel, requiring a pixel to be perceived as dirty several times before considering it as actually dirty (see Fig. 3.2e).

- *Ellipse representation:* Planners cannot cope with large state spaces, a small state is recommended to plan efficiently. Therefore, a new representation based on ellipses that fit the dirty areas is created. These ellipses provide a compact representation of the dirty areas shape and size (see Fig. 3.2f).



(a) Original image (b) Surface segmentation (c) Edges removal

(d) Background subtraction (e) Noise removal (f) Ellipses representation

Figure 3.2: Perception process.

Perception modules are initialized using a clean surface: the different parameters are adjusted to the minimum values for which the entire surface gets segmented as clean. Note that the subtraction thresholds would need to be adjusted again if either the surface or the lighting changes significantly, while all other parameters work with a wide range of surfaces.

This method focuses on selecting good cleaning strategies, and thus the perception system presented here is just a tool to test the cleaning skills. The proposed method is robust when cleaning a uniform surface with constant illumination, which was the only case tested in the experiments.

Actions

A set of actions is designed to clean a surface containing small objects like lentils, as shown in Fig. 3.3. The cleaning tool is always oriented perpendicular to the direction of motion to get effective moves. The robot starts and finishes actions a few centimeters away from the surface, so it does not push any lentils in between two actions. After reaching the starting point, the tool approaches the surface, performs the action while maintaining contact with it, and finally moves away from the surface as a preparation for the following action.

Actions are parametrized with the ellipses representing the dirty areas. To define the ellipses we use their positions and axes lengths and orientations.

One-arm cleaning actions

These actions move dirt to a container located close to an edge of the dirty surface. As they only have to push the dirt, one robotic arm is enough to execute them.

- `Fast move to container (ellipse)`:
  - *Starting point:* The point of the ellipse that is farthest from the container.
  - *Movement:* The robot moves the cloth to the container position, pushing the lentils towards it (Fig. 3.3a). It performs the shortest trajectory in joint space, which may not be completely straight.
- `Straight move to container (ellipse)`:
  - It is equivalent to *Fast move to container*, but uses path planning to ensure that the trajectory is straight. Although it is more precise, the addition of path planning makes it slower.

Two-arms cleaning actions

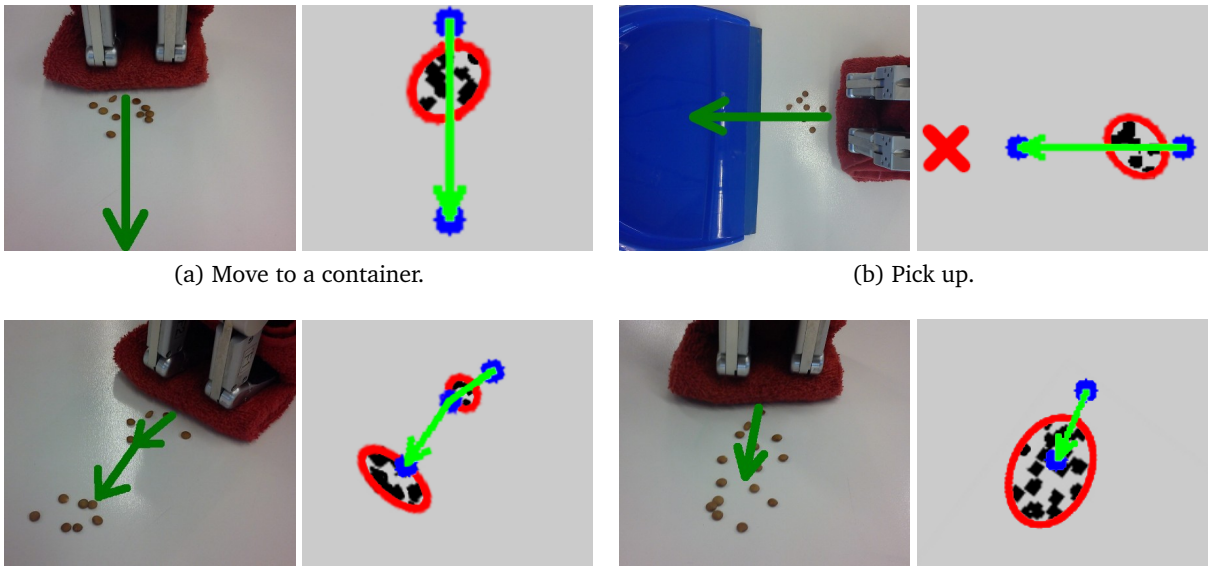
Having two arms allows the robot to execute pick up actions using a cloth and a dustpan. The arm with the cloth will push the dirt to the dustpan.

- `Pick up (ellipse)`: Assumes that the dustpan is grasped with the left hand, and the cloth with the right hand.
  - *Dustpan Starting Point:* Farthest point of the ellipse on the left.
  - *Cloth Starting Point:* Farthest point of the ellipse on the right.
  - *Movement:* The robot moves the cloth towards the dustpan position, pushing the lentils towards it (Fig. 3.3b).

Grouping actions

These actions rearrange the dirty areas on the surface, so that they become easier to clean by means of future actions.

- `Join 2 groups(ellipse1, ellipse2)`: The movement pushes the dirt of ellipse1 to ellipse2 (Fig. 3.3c).
  - *Starting Point:* The point of ellipse1 that is farthest from ellipse2.
  - *Second Point:* The point of ellipse1 that is nearest to ellipse2.
  - *Ending Point:* The point of ellipse2 that is nearest to ellipse1.



(a) Move to a container.

(b) Pick up.

(c) Join 2 groups.

(d) Grouping scattered dirt.

Figure 3.3: Cleaning actions. On the left we show the movements of the robotic arm grabbing a cloth. On the right, the robot movements are shown in the segmented 2D images. The arm with the cloth moves along the green arrow. In (b), the arm places the dustpan on the red cross.

- `Join 3 groups(ellipse1, ellipse2, ellipse3)`: Moves ellipse1 and ellipse2 to the position of ellipse3. The difference with executing twice the *Join 2 groups* action is that the robot arm continues smoothly the movement, not having to initialize the action again.
  - *Initial points:* The points of *Join 2 groups*.
  - *Continuation:* The point of ellipse2 that is nearest to ellipse3.
  - *Ending point:* The point of ellipse3 that is nearest to ellipse2.
- `Grouping scattered dirt(ellipse)`:
  - *Starting Point:* The ending point of the biggest axis of the ellipse.
  - *Movement:* The robot moves to the center of the ellipse, making a smaller and more manageable group of lentils in the process (Fig. 3.3d).

The important details to know about each action are its prerequisites, its effects and the time it takes to complete. These parameters can be obtained experimentally (Vaquero et al., 2013) or learned (Martínez et al., 2013). It is worth noting that all these actions are stochastic due to several factors:

1. **Perception errors:** Actions rely on the accuracy of the depth information provided by RGB-D cameras, which have a limited resolution. Although using a cloth provides some compliance, actions may fail to move the dirt as expected.

2. **Manipulation errors:** The same action may have different effects for similar dirty areas. For example, some lentils may spread during the trajectory in some cases, while they may move successfully in other similar situations.

3. **Tools:** Usual tools for cleaning, such as cloth, produce different results depending on the way they are grasped.

Planning

The planner selects the set of actions to execute based on the state and the rules. The task is quite complex, and selecting the fastest action sequence is challenging. For example, plans beginning with *grouping* actions may penalize in the beginning (remove no dirt) compared to *cleaning* actions, but they can provide the best results in the long run. Figure 3.4 shows a typical example of such behavior. Moreover, actions are stochastic, which makes the system more complex.

The problem is formulated as an MDP (see Sec. 2.1.3) where the planner has to obtain a sequence of actions to maximize $V^{\pi}(s)$. Afterwards these actions are converted into motions that the robot will perform to clean the dirty areas.

State representing the scene

The state $s = l_1, ..., l_n$ describes the dirty areas on the table. Each dirty area is an object, and the literals l_i represent the following:

- “small(?X)” indicates that the dirty area ?X is small.

- “medium(?X)” indicates that the dirty area ?X is medium.

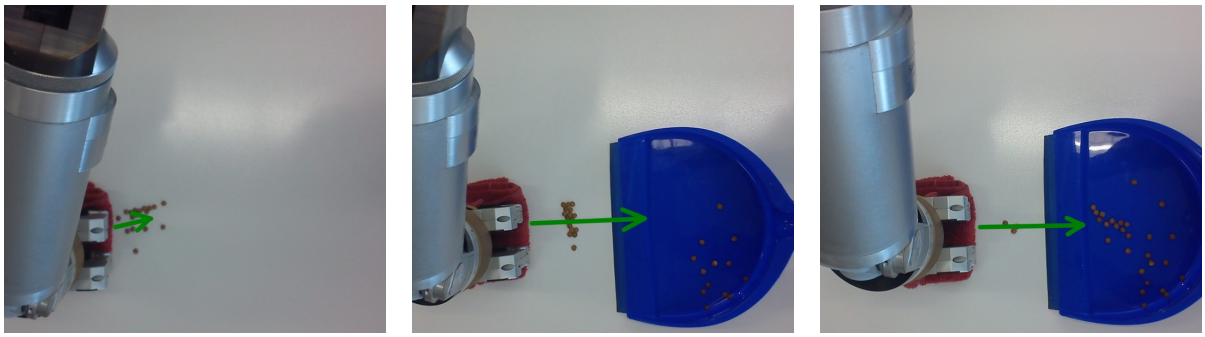
- “big(?X)” indicates that the dirty area ?X is big.

- “scattered(?X)” indicates that the dirt in ?X is scattered.

- “near(?X, ?Y)” indicates that the dirty areas ?X and ?Y are close to each other.

- “traversal(?X, ?Y, ?Z)” indicates that ?Z is positioned between ?X and ?Y. The *traversal* literal is used to indicate whether actions would also push other lentils across their trajectories.

(a) Initial state (b) Join groups on the left (c) Pick up right group



(d) Group together scattered lentils (e) Pick up lentils (f) Pick up remaining lentils

Figure 3.4: Planning example. The plan begins by joining the groups on the left (b) so that the robot will be able to pick them up later with just one action. After that, it continues by picking up the group on the right (c). Seeing that the group on the left is scattered, the robot executes a grouping action (d) and finally picks up the lentils (e). Nevertheless, the last action failed to clean a few lentils, so another pick up action (f) is executed to finish.

Perceptions are acquired while the robot is cleaning, so there may be some occlusions in the scene. The planner considers the state to be fully observable, and we will tackle the problem of occlusions later in Section 3.4.2.

Rules representing the actions

Cleaning actions are stochastic, i.e., they have different possible effects with associated probabilities. We model actions with Noisy Deictic Rules (NDR) (Sec. 2.1.1) that define their preconditions and effects. An example of a NDR rule is shown in Fig 3.5.

The rules can be obtained experimentally, executing each action a number of times until meaningful statistics are obtained to define the rules. However, the option of learning them autonomously is recommended, as it permits to adapt to possible changes in the robot or the environment.

Action:
pickUp(?X)
Preconditions:
dirt(?X), mediumSize(?X), ¬scattered(?X)
Effects (Success probability: literal changes):
0.4: ¬dirt(?X), clean(?X)
0.3: ¬mediumSize(?X), smallSize(?X)
0.2: ¬mediumSize(?X), smallSize(?X) sparse(?X)
0.1: noise

Figure 3.5: NDR rule example for picking up lentils.

Planning with uncertain actions

In this section we introduce the planner and some design considerations needed to tackle stochastic domains.

Using a probabilistic planner is important when actions have several possible effects with different probabilities. Deterministic planners and replanners only consider the most probable effect for each action, while a probabilistic planner takes into account all effects (Yoon et al., 2007). For example, consider that there are two scattered dirty areas and two cleaning actions.

- The first action cleans a scattered dirty area with probability 0.5 and a compact one with probability 0.8.
- The second action joins two dirty areas in a scattered group with probability 0.3, and joins them in a compact group with a probability 0.2.

The best plan joins the two groups and then cleans them. It has a $0.3 * 0.5(scattered) + 0.2 * 0.8(compact) = 0.31$ probability of cleaning both. In contrast, a deterministic planner would choose the first cleaning action twice as it considers just the best effect, and both groups would be cleaned with probability $0.5 * 0.5 = 0.25$.

Probabilistic planner

The system uses PRADA (Lang and Toussaint, 2010), a model-based planner for complex domains. Defining s^i as the state at time i, a^i as the action to be executed, and $\#dirt(s^i)$ as the number of dirty areas in state s^i, the sum of expected rewards $V(s, \overline{a})$ of a sequence of actions is obtained as follows:

$$V(s^0, a^{0:T-1}) = \sum_{t=0}^{T} \gamma^t P(\#dirt(s^t) - \#dirt(s^{t+1}) \mid a^{0:t-1}, s^0) \tag{3.1}$$

where $P(\#dirt(s^t) - \#dirt(s^{t+1}) \mid a^{0:t-1}, s^0)$ is the probability that a number of dirty areas are cleaned at time t.

The reward increases with the number of dirty areas cleaned, but also, as we want to clean as fast as possible, the discount factor γ makes the reward for cleaning an area decrease with the number of actions previously executed. This way, the best rewards will be obtained by plans that clean many areas fast, which are the best ones for our task.

Issues

Obtaining effective plans in real-world applications can be complex. The planner should take a limited amount of time while providing good results and adapting to all the contingencies.

Computational time

PRADA is a suboptimal planner, so the probability of finding good plans depends on the fraction of actions sampled. The maximum sampling number n should be proportional to the complexity of the problem, which depends on the number of dirty areas $\#dirt(s)$, the number of available actions $|A|$, and the plan length H, which is the maximum number of actions in the plan. The coverage of the state space is

$$coverage = \frac{n}{(\#dirt(s) \cdot |A|)^H} \tag{3.2}$$

and it should be at least significant enough to find good plans.

If the planner samples only a small portion of the state space there is a high probability that it will not find the best plans, but increasing the number of samples also increases the computational cost. For difficult problems two approaches can be taken to get good plans: increasing the number of samples, or reducing the problem complexity by ignoring some of the dirty areas.

It should be noted that it is not worth using a planner if more time is spent in planning than is saved by applying the plans. Therefore a maximum sampling number was set so the planner never spent more than a few seconds, and this number was reduced when few dirty areas were present on the surface.

Plan length

The planner maximizes the results obtained for a given plan length that has to be specified before planning. The plan length should be large enough to permit cleaning all the surface, but not so long that the last actions become useless while increasing the planning time. Good empirical results were obtained with a plan length of $1 + \#dirt(s)$, having one action for each dirty area and an extra action.

Replanning

Actions are stochastic, so plans will not always perform as expected. After a plan is executed, a new plan is generated based on the updated state containing only the remaining dirty areas. The robot will continue generating and executing new plans until the surface gets cleaned completely.

Moreover, to minimize cleaning time, the system is continuously taking perceptions in parallel while the robot cleans. The planner considers the state to be fully observable, but the robot arm may occlude some parts of the scene, so once the robot has cleaned the observable surface, it will take the arm out of the field of vision to check if there are remaining dirty areas.

Partial Plans

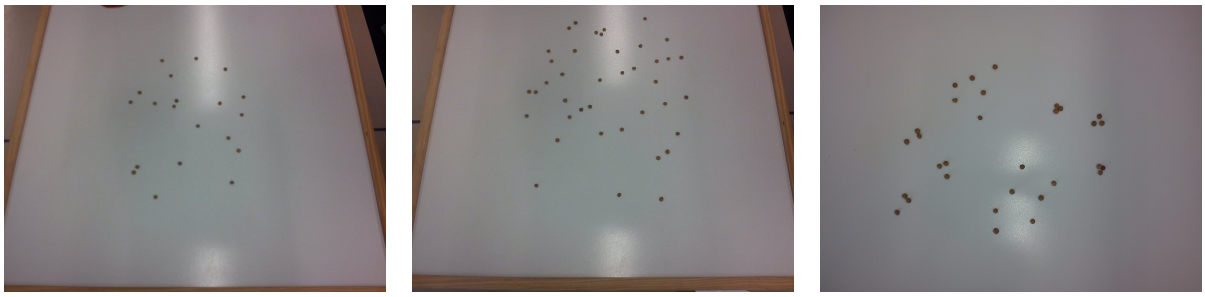
Executing only partial plans can also improve the results. Actions have many possible effects, which makes it hard to predict the state after executing a part of the plan. As later actions may be too uncertain, it is a good idea to execute only the first actions of a plan before generating a new plan with updated perceptions. Newer plans will be more precise and get better results. Experimental validation of this idea is presented in Section 3.5.

Experimental results

The experiments were performed on a surface with many lentils arranged as seen in Fig. 3.6. The simpler layouts had just two groups of lentils, while more difficult ones had up to 40 lentils scattered all over the surface. The core of the experiments was carried out using a WAM arm and a RGB-D camera attached to the environment (the hand-eye calibration parameters were obtained using standard methods). The REEM robot has been used to validate the results on a different robotic platform, but no systematic experiments were done that permit to obtain meaningful statistics about its performance. Videos of the experiments, including also the cleaning of a whiteboard, can be found at `http://www.iri.upc.edu/groups/perception/surfaceCleaning`.

Two different cleaning skills are validated here. The first one consists in moving the lentils with a cloth to a container positioned near an edge of the surface, and the second one consists in picking up lentils with the help of a dustpan. The γ parameter of the planner was set to 0.95 which led to good results in our tests. The sampling number n was set taking into account the most difficult scenarios, and a value of 10^4 was enough to obtain good results while spending at most ~ 4 seconds planning.

(a) 2 groups of 10 lentils (b) 4 groups of 5 lentils (c) 8 groups of 5 lentils

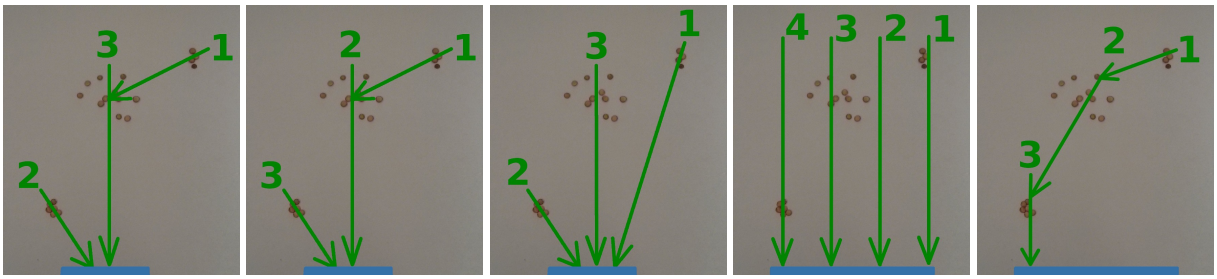


(d) 20 lentils scattered (e) 40 lentils scattered (f) 30 lentils scattered + groups

Figure 3.6: Experiments setup.

To the best of our knowledge, there are not approaches directly comparable to ours, because each tackles a slightly different cleaning problem. The closest is that of Hess et al. (2013), which nevertheless uses an accurate vacuum cleaner instead of an inaccurate cleaning tool such as our rag. We compared different configurations of our approach with a reactive method, a fixed program and the approach by Hess et al. (2013), whose examples are shown in Fig.3.7.

- Planning and executing 1, 2 or 3 actions before replanning. Using 4 or more actions led to worse results as estimating the layout of the lentils becomes very difficult.
- A reactive strategy that uses the segmentation into groups of lentils. The robot starts cleaning the dirty area that is farthest from the goal, and continues cleaning the dirty areas that are closest to the arm position.
- A fixed program to wipe the bounding box of the dirty areas using a zig-zag like motion. The robot wipes in straight lines from right to left across the table.
- Hess et al. (2013) cleaning skills using a rag instead of a vacuum cleaner. This approach uses a TSP planner to obtain the shortest path that passes through all dirty areas (grouping them in a single dirty area) and finally the dirt is cleaned. This policy yielded good results when using a vacuum cleaner (Hess et al., 2013), but using a rag is more challenging since grouping all dirt together and then cleaning it involves a lot of uncertainty.



(a) Plan, 1 action (b) Plan, 2 actions (c) Reactive (d) Complete wipe (e) Hess et al.

Figure 3.7: Executions with different strategies in the task of moving lentils to a container. **Plan and execute 1 action:** the robot joins the two upper groups together. Then it replans while having the arm occluding the upper group, so it chooses to clean the dirty area on the bottom left. Finally it replans and cleans the upper group. **Planning and executing 2 actions:** The robot joins the two upper groups and cleans them. Then it replans and cleans the last group. **Reactive:** Cleans one group after another without planning. **Complete wipe:** The robot wipes in straight lines to clean the bounding box of the dirty areas without planning. **Hess et al. (2013):** The robot groups all dirty areas following the shortest path, and finally cleans them.

Moving lentils to a container

This task consisted in dragging the lentils laying on a table to a container positioned near the edge of the table. The robotic arm used a cloth to push the lentils to the container.

In Figure 3.8-left we show the results obtained when cleaning the different layouts of lentils (Fig. 3.6). The proposed approach was applied until all lentils were cleaned. In this experiment, planning gives similar or sightly better results in simpler tasks, and large improvements in difficult tasks. The reactive method scales very bad, as it cleans dirty areas one by one without grouping them. Hess et al. (2013) approach was devised to use an accurate vacuum cleaner, but adapts badly to unexpected problems that arise when using uncertain actions such as moving dirt with a rag. In contrast, planning adapts to failed actions, and avoids joining groups that are too big and far from each other. When planning, overall, executing two actions before re-planning usually yields the best results. A lot of re-planning is required when executing just one action, while the prediction of the state becomes too poor when using more than two actions in difficult problems.

Figure 3.8-right shows the time spent in experiments entailing moving 40 scattered lentils to a container, which was the most difficult problem instance. Planning and perception time (which includes the time moving the arm away from the scene to tackle possible occlusions) constitute only a small overhead compared to cleaning actions. Perception time is also smaller when executing longer sequences, as the robot tries to clean more dirty areas before using newer perceptions.

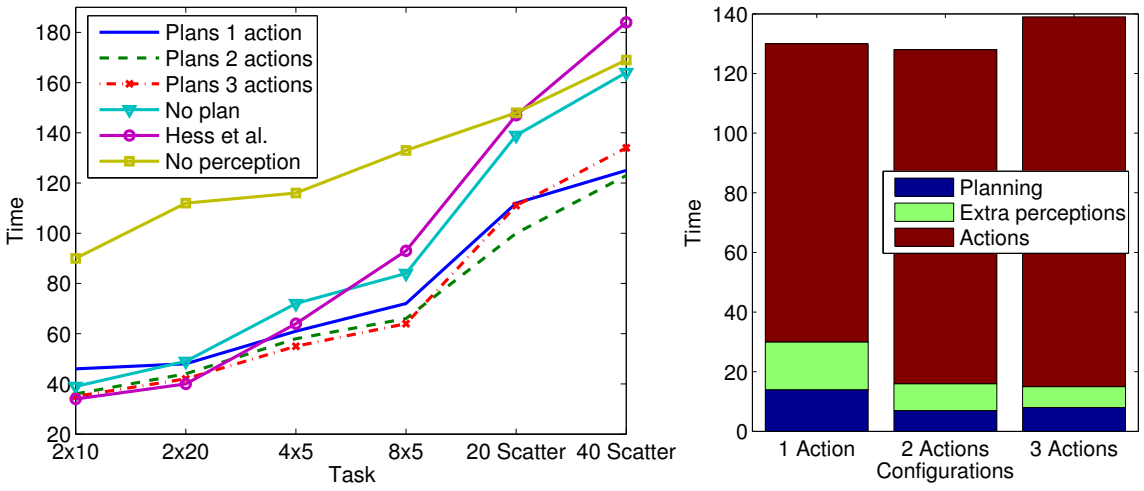


Figure 3.8: **Left:** skill of moving lentils to a container. The results shown are the mean over 5 runs. **Right:** time for the case of moving 40 scattered lentils to a container. Extra perceptions include the movements and perceptions needed to tackle occlusions by taking the arm out of the field of view (see Section 3.4.2).

Picking up lentils

In this experiment lentils are picked up from a table using two robotic arms grabbing a cloth and a dustpan. The arm with the cloth performs the grouping actions and also pushes lentils into the dustpan, while the arm with the dustpan just has to place it near the lentils that are going to be picked up.

In Figure 3.9 we show the results obtained when picking the different layouts of lentils. The robot kept cleaning until all lentils were picked up. In the simplest cases of cleaning two groups, the planner gives similar results to the reactive method, but as the complexity of the problem increases, it can be seen that using a planner improves the results significantly. The planner takes advantage of joining lentils in a few compact groups before picking them up. The action to pick up lentils is the slowest, so decreasing the number of times it has to be executed leads to better results. In general, executing 2 actions before replanning provides the best results, but adding a third action slightly enhances the results in some situations.

Conclusions

In this chapter we have shown the use of a planner in a real robotic system, and the improvements that it provides over a reactive strategy and a fixed program. The planner was used in the task of cleaning surfaces with a set-up consisting of a RGB-D camera and robot arms without force control.

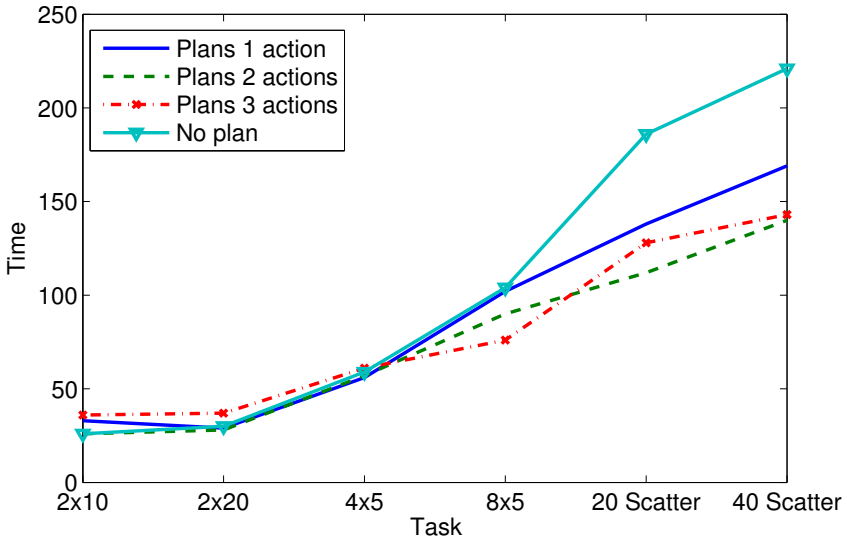


Figure 3.9: Skill of picking up lentils. The results shown are the mean over 3 runs.

Real world tasks are usually stochastic and quite complex. Thus, selecting sequences of actions is very difficult in this kind of domains. Using a planner has been very useful to get good solutions to accomplish these tasks faster. The planner takes into account the different probabilities of the action effects and obtains good plans to optimize results. In simple scenarios the presented method performs similar to reactive strategies, but in complex scenarios the ability to plan proves to be crucial to improve results.

As we can tackle stochastic actions, we can also decrease the number of constraints of the system. In this work we have performed actions that maintain contact using a RGB-D camera instead of the common approach of force control. The depth information of RGB-D cameras and the compliance of using a cloth to clean, gives good results, but still adds some uncertainty in the actions. This extra uncertainty can be handled as we are planning with stochastic actions, and thus the constraint of having force sensors can be removed, allowing us to apply the method to the robot REEM.

4

Reinforcement Learning with Active learning

Model-based reinforcement learning is a powerful paradigm for learning tasks in robotics. However, in-depth exploration is usually required and the actions have to be known in advance. Thus, we propose new algorithms that integrate the option of requesting teacher demonstrations to learn new domains with fewer action executions and no previous knowledge. Demonstrations allow new actions to be learned and they greatly reduce the amount of exploration required, but they are only requested when they are expected to yield a significant improvement because the teacher's time is considered to be more valuable than the robot's time.

After describing previous work, in Sec. 4.3 we will present the REX-D algorithm that extends the REX algorithm (Lang et al., 2012) to include a teacher. Combining the relational generalizations of REX and demonstration requests, REX-D reduces the amount of exploration required by up to 60% in some domains, and improves the success ratio by 35% in other domains. Moreover, as a planning model is being used, we show how model analysis can be used to provide guidance to the teacher to get better demonstrations. Model analysis also reduces the number of demonstrations by finding subgoals and rule alternatives. The REX-D algorithm has been published in (Martínez et al., 2016a, 2014a).

Section 4.4 describes an extension of REX-D, in which the decision maker reasons about dead-ends and their causes. Some such causes may be dangerous action effects, i.e., effects leading to unrecoverable errors if the action were executed in the given state. The method allows the decision maker to skip the exploration of risky actions and guarantees the safety of planned actions. If a plan might lead to a dead-end (e.g., one that includes a dangerous action effect), the decision maker tries to find an alternative safe plan and, if not found, it actively asks a teacher whether the risky action should be executed. This work has appeared in (Martínez et al., 2014b, 2015b).

Finally, Sec. 4.5 presents the V-MIN algorithm, which is a generalization of the REX-D algorithm: V-MIN is not limited to goal-driven tasks, and allows the user to control the amount

of teacher requests that should be requested. V-MIN requests demonstrations from a teacher whenever a value higher than a given threshold V_{min} cannot be obtained. This threshold sets the degree of quality with which the agent is expected to complete the task, thus allowing the user to either opt for very good policies that require many learning experiences, or to be more permissive with sub-optimal policies that are easier to learn. The threshold can also be increased on-line to force the system to improve its policies until the desired behavior is obtained. The V-MIN algorithm has been published in (Martínez et al., 2015c).

Note that all the algorithms presented in this section assume that the models consist of NDR rules (Sec. 2.1.1). Nevertheless, planning operators (Sec. 2.1.1) could be used instead by changing all instances of $eff(r)$ with $head(o)$ and $pre(r)$ with $body(o)$, where r is a NDR rule and o is a planning operator.

Previous work

RL has proven to be an excellent tool to learn and solve a task that is unknown initially (Littman, 2015). More precisely, in tasks where the decision maker has little input data and long periods of time to process it, model-based RL is the correct approach as it obtains the most information from every experience. Furthermore, if a model for planning is learned, we can get all the advantages of using automated planners: the same model can be used with different states and reward functions, and new actions can be added easily.

To learn tasks as rapidly as possible, we need a highly compact representation of the model, and thus we use relational models. These models generalize over different objects of the same type, thereby reducing the learning complexity of domains to the number of different types of objects in them. Several approaches have applied successfully RL using relational models (Cocora et al., 2006; Katz et al., 2008; Rodrigues et al., 2011; Džeroski et al., 2001). There are several RL approaches that can learn complete relational models for planning (Diuk et al., 2008; Li et al., 2011; Walsh et al., 2009), but they require a large amount of samples to make sure that they can solve the problem (Walsh, 2010). Other approaches use heuristics to reduce the number of samples needed to make the problem tractable, such as TEXPLORE (Hester and Stone, 2013) and REX (Lang et al., 2012). Specifically, REX uses Pasula et al. (2007)'s model learner internally, and proposes an exploration method based on R-MAX (Brafman and Tennenholtz, 2003) and E^3 (Kearns and Singh, 2002) that takes advantage of the relational representation to generalize and reduce the exploration required to learn relational models. However, heuristic learners may obtain models that are a local minimum as they cut exploration.

The algorithms presented in this chapter, REX-D and V-MIN, combine reinforcement and active learning to further reduce the number of samples required to solve a task. Moreover, as

will be explained later, these algorithms can learn new actions by requesting demonstrations, which makes the adaptation to new tasks easier.

The integration of teacher demonstrations has already been tackled within RL-like algorithms to improve the learning process. Meriçli et al. (2012) presented an algorithm where the teacher issues corrective demonstrations if the robot does not perform as desired. In the approach by Walsh et al. (2010), the teacher reviews the planned actions, and whenever they are not optimal, a demonstration is shown. In these approaches the teacher has to intervene to improve the agent's behavior. In comparison, our algorithms actively request demonstrations when needed, releasing the teacher from having to monitor the system continuously.

Active demonstration requests have been included in algorithms with confidence thresholds (Grollman and Jenkins, 2007; Chernova and Veloso, 2009), which request demonstrations for a specific part of the state space whenever the system is not sure about the expected behavior. However, they only learn from demonstrations, and as teacher time is considered to be very valuable, demonstration requests should be limited and replaced with exploration whenever possible. Walsh and Littman (2008) propose a pessimistic approach where every literal is a precondition of a effect unless proven otherwise, so many demonstrations have to be requested until a general model can be obtained.

Finally, there are also approaches that request demonstrations from the teacher when the planner cannot find a solution with its current set of rules, so they learn by combining demonstration requests and exploratory actions. Agostini et al. (2016) take an optimistic approach in which they start with general rules and specialize them as counterexamples are found. However, this approach does not include relational generalizations and cannot tackle probabilistic domains.

Related RL Algorithms

Below we present three RL algorithms in which our proposals will be based: R-MAX, E^3 and REX.

The R-MAX algorithm

R-MAX is a RL Algorithm (Brafman and Tennenholtz, 2003) that takes an "optimism in the face of uncertainty" approach to explore. It uses a threshold ζ that is the number of times that a state-action pair has to be visited before considering it as "known". R-MAX creates a new MDP M_{R-MAX} where all unknown state-action pairs give a maximum reward, so the agent will implicitly visit under-explored areas when planning in the MDP.

If the transition model T can be efficiently KWIK-learned, then R-MAX has been proven to be PAC-MDP (Strehl et al., 2009), which means that it learns near-optimal behavior in MDPs with a polynomial number of samples $\widetilde{O}(|S|^2|A|/(\epsilon^3(1-\gamma)^6))$ where ϵ is an accuracy parameter.

The E3 algorithm

The E^3 (Kearns and Singh, 2002) algorithm is also based on the concept of known states: a state is considered to be known when all the state-action pairs for that state have been visited at least the confidence threshold ζ number of times. Depending on the experiences of the algorithm, it proceeds as follows.

- Whenever the agent enters an unknown state, it performs the action that requires the fewest times to explore.

- If it enters a known state, and a valid plan through known states is found, then that plan is executed (exploitation).

- If it enters a known state but no plan is found, it plans using an MDP where unknown states are assumed to have a very high value (planned exploration).

The REX algorithm

Lang et al. (2012) proposed a solution to the exploration-exploitation dilemma in relational worlds, which employs the relational representation to reduce the number of samples before treating a state as known. Their proposed method uses the following context-based density formula for state-action pairs:

$$k(s,a) = \sum_{o \in \mathcal{O}} |E(o)| I(o = o_{s,a}), \tag{4.1}$$

where $|E(o)|$ counts the number of experiences that cover the operator o with any grounding and $I(\cdot)$ is a function, which is 1 if the argument is evaluated as true and 0 otherwise.

The REX algorithm uses Eq. 4.1 to learn with much fewer action executions than previous approaches in domains where many objects of the same type appear. It has two variants: one based on R-MAX and another based on E^3. The REX algorithm generalizes R-MAX and E^3 to a relational domain, but it maintains the same concepts. The R-MAX variant of REX has been proven to be PAC-MDP (Lang et al., 2012) (learns near-optimal behavior in MDPs with a polynomial number of samples).

REX-D

The REX algorithm was devised to apply the context-based density function in Eq. 4.1 to E^3 and R-MAX, improving the generalization over different states. Although these generalizations improve the learning performance, they also have drawbacks. Using a context-based count function implies that not all states are explored before considering them as known, as it assumes that all states within a context behave likewise. Therefore, two or more contexts may be learned as one, and the RL algorithm will not realize that they are two different contexts (i.e. rules) until state-action pairs of both contexts are executed. To avoid this problem a large amount of exploration actions may be required so that all the needed contexts are found.

In this section we propose the REX-D algorithm, which combines reinforcement and active learning to solve initially unknown task with fewer action executions. REX-D can find all the contexts that are needed to complete the task with just a few teacher demonstrations, in contrast with the large number of actions that REX would require. Moreover, REX-D provides another important advantage, it can learn new actions as they are required through demonstrations.

After introducing REX-D, we will show that the learned model can be analyzed to provide guidance to the teacher so that the best demonstrations are received, and also to find alternative models when no successful plan can be found.

Algorithm

We propose the algorithm REX-D (Algorithm 1), where we modify the E^3 variant of the REX algorithm (Sec. 4.2) by adding the option to request demonstrations from a teacher. First, REX-D explores the list of available actions until they are considered to be known in the same manner as REX (Eq. 4.1). Once in a known state, if the planner can find a plan then an action is executed. However, if no solution is found, a demonstration is requested to the teacher. The treatment of the demonstration depends on the current knowledge of the agent, as follows.

- If the demonstration is the first execution of the action, the agent learns how to execute that action and adds it to the set of actions that the learner must learn.

- If the action has already been executed, the model is refined by the learner.

The ability to make a few requests of a teacher inside the RL loop has two main advantages: faster learning and the ability to add actions as they are required.

Algorithm 1 REX-D

Input: Reward function $\mathcal{R}$, confidence threshold ζ

Set of experiences $E = \phi$
Observed state s_0
loop
    Update transition model T according to E
    if $\forall a \in A : k(s, a) \geq \zeta$ **then** ▷ *If the state is known*
        Plan using T
        if a plan is obtained **then**
            $a_t =$ first action of the plan
        else
            Request demonstration
            $a_t =$ action demonstrated
        end if
    else
        $a_t = argmin_a k(s_t, a)$
    end if
    Execute a_t
    Observe new state s_{t+1}
    Add $\{(s_t, a_t, s_{t+1})\}$ to E
end loop

Faster learning

Executing actions takes a long time for robots, so it is recommended to request help from the teacher to save large periods of learning time.

Exploration is performed until a known state is reached. However, once a known state is entered, if no plan can be found, the decision maker has to decide where to continue exploring. Exploring the whole state space requires a large number of samples; thus, a demonstration may be requested to guide the system through the optimal path.

The problem when learning a model only by exploration (in the case where a list of actions is available) is finding examples where the actions are successful. As noted by Walsh (2010), when an action fails because its preconditions are not satisfied (which we will call *negative examples*), these experiences are highly uninformative because the learner cannot determine the reason for the failure. However, *positive examples* where an action changes literals, provide very useful information because a superset of the literals that belong to the precondition is identified.

Proposition 4.1. In the worst case, learning the preconditions of a NDR rule r needs $O(2^{|L_r|})$ examples, where $|L_r| = \sum_{l \in L} |terms(r)|^{|terms(l)|}$ is the number of literals that may appear in the rule preconditions.

Proof. If the rule preconditions have a literal set L_r which is either positive or negative, then only one combination that uses all the literals satisfies the preconditions, and $2^{|L_r|}$ action executions are needed to test all possible combinations. In the worst case, the last combination tested is the only valid combination, thus $2^{|L_r|}$ examples are needed to find it.

Example Consider that we have to open a box with the action *open(X,Y)*, but it requires that the objects X and Y are connected in a particular way (using the literals *connected(X,Y)* and *on(X)*). In this case, $|L_r| = 2^2 + 2^1 = 6$, and the possible preconditions are:

$$\pm connected(X,Y), \pm connected(Y,X), \pm connected(X,X), \pm connected(Y,Y), \pm on(X), \pm on(Y).$$

In total, there are: $2^{|L_r|} = 2^6 = 64$ combinations. In the worst case, where the correct combination is the last one tested, the $open(X,Y)$ action has to be executed in 64 different states until a suitable one is found. □

Proposition 4.2. With only one positive example $e = (s, a_r, s')$, the action rule preconditions can be learned with $O(|s|)$ examples.

Proof. Because the preconditions comprise a conjunction of literals, we know that only the literals in the state s where the action was executed may be present in the rule preconditions. Thus, setting each one of these literals to its opposite to test whether it is actually one of the preconditions requires $|s|$ examples. □

Proposition 4.3. Adding a new positive examples of a rule $E_r = \{e_1, ..., e_n\}$ reduces the complexity of learning its preconditions to $O(|L_{E_r}|)$ examples, where $L_{E_r} = \{s_1 \cap ... \cap s_n\}$.

In real-world scenarios, learning rules requires far fewer experiences than the worst cases described above. In general, most literals are irrelevant to the execution of an action, which increases the probability of obtaining a positive example. Furthermore, robots operate in small subsets of the overall state space and their actions are designed to be executed only in the parts of the state space that they can reach. Although actions can be learned with fewer experiences, the previously described propositions provide a general view of the complexity of the learning process and the advantages of using demonstrations.

Online action additions

Initially, the learning system does not know which actions are available because the teacher has to demonstrate them only when they are needed. Therefore, exploration alone is not a valid strategy because a previously unknown action may be required to complete the task. When the

current state is known (using only the list of actions that have already been learned) and no plan is found, a demonstration is requested by REX-D. The teacher will execute the best action for that state, which may be a new and previously unlearned action.

Generalizing to different tasks

A model is generated to represent the actions required to complete a task. If these actions are also used in another task, previous experiences are useful for this new task. Moreover, relational representations allow generalization over different objects of the same type, and thus new tasks that use the same types of objects but different layouts will also reuse previous models.

In addition to transferring the model to other tasks, the ability to request demonstrations from the teacher gives the system much more flexibility. Thus, actions do not have to be defined in advance, but the system will request demonstrations from the teacher when they are needed if unknown actions are required during the execution.

Using Taylor and Stone (2009)'s transfer learning categorization, the new task can change the start state, the goal state, and the number of objects without requiring further learning. Moreover, new actions can be incorporated through demonstrations, and the reward function can change without having to modify the rest of the model.

Note that the state transitions in the initial task must be also valid for the new task. If those transitions were not valid, a map from one domain to the other could be used to speed up the learning process (Bianchi et al., 2015).

Teacher Guidance

The REX-D algorithm requests demonstrations from the teacher whenever it does not know how to complete the task. However, several actions may be required to complete the task, only one of which might be unknown to the system. Thus, if no guidance is provided, the teacher may demonstrate actions that the system already knows. Our objective is to avoid these unnecessary demonstrations, thereby minimizing the number of interactions with the teacher.

Figure 4.1 shows an example of the Cranfield benchmark scenario (as explained in 2.2.2), where the robot has to mount the different parts to complete an assembly. In this example, the robot does not know how to place a pendulum, so it cannot find a plan that reaches the goal and a demonstration is requested. However, the teacher does not know what is the problem, and two possible actions are equally good: placing a round peg (red), or placing the pendulum (gray). The guidance that we propose in this section would warn that the "pendulumPlaced" predicate cannot be obtained. With that information, the teacher will know that it should teach the action to place the pendulum.

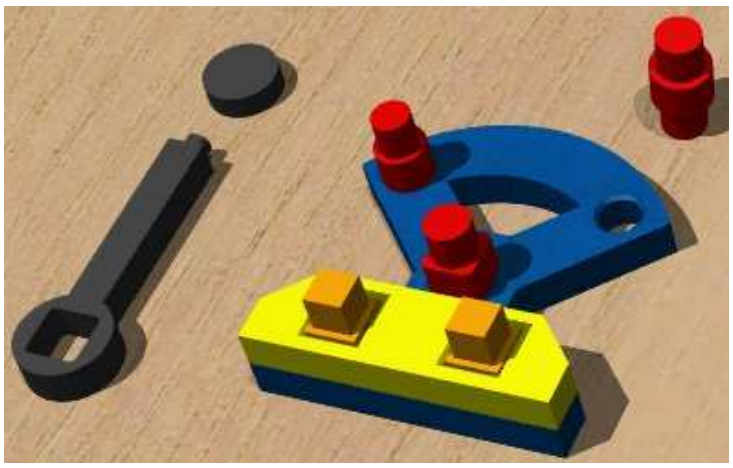


Figure 4.1: Intermediate state in the Cranfield benchmark. The pendulum, a rounded peg, and the front faceplate still have to be placed.

Obtaining guidance from planning excuses

The excuses presented in Sec. 2.1.7 are used to find problems in the model to guide the teacher so he demonstrates the action that the system needs. There are several possible reasons why the system may fail to plan, i.e., the wrong preconditions have been added, missing action effects, or dead ends.

Proposition 4.4. Let R be the complete rule set where there is an incorrect rule $r_{incorrect} \in R$, which has l_w as a precondition but its effects are actually not dependent on it. Let φ be an excuse that adds l_w to the state and if φ does not prevent any other transition from other rules, φ is an acceptable excuse.

Proposition 4.5. Let R' be an incomplete rule set, which has one rule missing, $r_{missing} \notin R'$. If the effects *eff*$(r_{missing})$ are required to achieve the goal and no other rule can obtain the literals, i.e., $\exists l \in$ *eff*$(r_{missing}) \mid \forall r' \in R', l \notin$ *eff*(r'), then an excuse φ that includes the necessary changes to *eff*$(r_{missing})$ is an acceptable excuse.

We will suppose that excuses found point at the problems explained in the previous propositions. However it should be noted that excuses that create invalid alternative paths to the goal may also exist in some domains. The teacher will be warned about possible wrong preconditions or unknown needed effects:

- Warning about incorrect preconditions. When the changes in the excuse φ affect the preconditions of one or more rules ($\exists r \in R \mid \varphi \in pre(r)$), the teacher is warned that these rules require the excuse φ. If one of the preconditions is incorrect, the teacher can simply execute the action to prove to the system that it is not actually required.

- Warning about missing effects. The teacher is warned that the system does not know how to obtain the changed literals in φ.

Guidance examples

It is recommended that the predicates used in the domain have descriptive names to make excuses guidance useful for non-technical teachers. Teaching the system can be relatively easy if the domain semantics are described appropriately in the predicates, as demonstrated by the following examples of robot messages.

- *I want to "pickUp(block)", but this requires "onTable(block)" [possible incorrect precondition]*. If the teacher knows that the "onTable" predicate is not required, he only has to execute the "pickUp" action to demonstrate that the "onTable" predicate was not required.

- *I don't know how to remove a "flatTire()" [missing effect]*. The action "changeTire" that removes "flatTire" has to be demonstrated.

- *I don't know how to remove a "broken(table)" [missing effect]*. If the table is needed to complete the task and it cannot be repaired, this is actually a dead end. Thus, the teacher has to signal a dead end to REX-D.

Improving Learned Models to Minimize Teacher Interactions

The model learners used (Sec. 2.1.4) obtain a model that maximizes a score function, but it may not be the correct model. In this section we show that we can use planning excuses (Sec. 2.1.7) to look for alternative better models. Moreover, we also show that subgoals permit taking the agent to the state where a demonstration is actually needed.

Rule alternatives

The system uses a greedy heuristic learner for obtaining a model that explains previous experiences (Sec. 2.1.4). If these experiences are not sufficiently complete, as the learned models optimize a score function (Eq. 2.8 or Eq. 5.2), many different models explain the experiences with similar scores.

The ζ parameter (for considering known states and actions, as introduced in Sec. 4.3.1) has to be optimized for each different domain to obtain good results. However, we use a small threshold to learn rapidly, and after that excuses are used to generate alternative models that compensate for this overconfidence.

Algorithm 2 GenerateRuleAlternatives

Input: Current rule set R, state s, experiences E, valid excuses $\varphi_1 ... \varphi_n$
Output: New rule set R'

1: $R_{candidates} = \phi$ ▷ *List of rule set candidates*
2: **for all** $\varphi_i \in \varphi_1 ... \varphi_n$ **do**
3: $l_\varphi = s \triangle \varphi_i$ ▷ *Literals changed by the excuse*
4: $R_\varphi =$ GetRulesRequiringExcuse(R, s, l_φ)
5: **for all** $r_\varphi \in R_\varphi$ **do**
6: **for all** $l_i \in s$ **do**
7: $R_{new} = R$
8: Substitute precondition l_φ for l_i in rule $r_\varphi \in R_{new}$
9: Add R_{new} to $R_{candidates}$
10: **end for**
11: **end for**
12: **end for**
13: $R' = R$
14: **for all** $R_i \in R_{candidates}$ **do**
15: **if** Score$(R_i) \simeq$ Score(R') **then** ▷ *Score() is computed using Eq. 2.8*
16: **if** $\exists$ plan(R_i, s) **then**
17: $R' = R_i$
18: **end if**
19: **end if**
20: **end for**

Let E be a set of experiences and R a rule set that explains those experiences. If E does not cover the whole state space, different rule sets R' may explain the experiences as well as R (score$(R) \simeq$ score(R')). If no plan is found but the system obtains an excuse φ, it is hypothesized that R might not be the correct rule set. The changes in the excuses $l_\varphi \in s \triangle s_\varphi$ are used to find alternative models. The system will analyze rules with excuse changes in their preconditions $l_\varphi \in pre(r)$ and check for alternative equivalent rules that may obtain a plan for the current state. Note that the complexity grows exponentially when finding rule alternatives with excuses that change more than one literal.

When no plan is found, the algorithm `GenerateRuleAlternatives` (Algorithm 2) tries to find an alternative rule set that explains past experiences and obtains a plan from the current state.

- First, for each excuse that explains the planner's failure, it finds the rules that are required to reach the goal and that can only be executed when the excuse has been applied. The `GetRulesRequiringExcuse` routine (Algorithm 3) finds these rules and it outputs the set R_φ of rules that allow a valid plan to be obtained by removing the literals changed by the excuse from their preconditions.

Algorithm 3 GetRulesRequiringExcuse

Input: Current rule set R, state s, excuse changes l_φ
Output: Rules R_φ that obtain a plan if l_φ is removed from preconditions

$R_\varphi = \phi$
$R_{pre_\varphi} = \{r \mid r \in R, l_\varphi \in pre(r)\}$. ▷ *rules with l_φ as a precondition*
for all $r_i \in R_{pre_\varphi}$ **do**
    $R'_i = R$
    $r'_i = r_i$
    Remove l_φ from $pre(r'_i)$
    Substitute r_i for r'_i in R'_i
    if $\exists$ plan(R'_i, s) **then**
        Add r_i to R_φ
    end if
end for

- R_φ is used to create the set of candidates that can be valid rule alternatives. Each candidate is generated by making a copy of the current rules R_{new} and modifying one of the rules $r_\varphi \in R_{new}$, which is also in R_φ. The preconditions r_φ are changed by replacing a literal that is changed by the excuse φ with a literal that is present in the current state $l \in s$. The aim is to check whether the literals changed by the excuse are actually in the preconditions, or if any other literals in the current state should be used instead.

- After the rule set candidates are obtained, the score is calculated (using Eq. 2.8 or Eq. 5.2) for each. If the score is at least equal to the current rule set score, then the candidate can explain past experiences as well as the current rule set, and thus it is a good candidate. Note that although many candidates are usually generated, the score can be calculated very rapidly and most of the candidates are pruned during this step.

- Finally, the planner uses good candidates to plan from the current state and a candidate is selected as the output rule set if it yields a plan.

Subgoals

Help is requested from a teacher if the planner cannot obtain a sequence of actions to complete the task. However, the teacher may be asked to perform an action that has to be executed after executing other actions that the system already knows. In some scenarios, it is useful to get as close as possible to the goal to minimize the number of actions that the teacher has to execute. Thus, subgoals are used to approach the goal. Note that the goal has to be defined as a set of target literals for this feature to work.

Algorithm 4 SubgoalGeneration

Input: Goal G, excuse changes l_φ, causal graph CG_R
Output: Subgoal G'

1: $G' = G$
2: **while** $\exists l \in G' \mid l_\varphi \in requirements(l)$ **do**
3: Remove $\{l \mid l \in G', l_\varphi \in requirements(l)\}$ from G' ▷ *Remove literals that depend on* l_φ
4: Add $\{l' \mid l \in G', \exists \text{ arc}(l', l) \in CG_R\}$ to G' ▷ *Add the requirements for the removed literals*
5: **end while**

Definition 4.1. (Subgoal) Given a goal $G = \{l_1, ..., l_n\}$, a subgoal is a set of literals that belongs to the goal, or the requirements of goal literals $G' = \{l_1, ..., l_n \mid l_i \in G \vee l_i \in requirements(G)\}$. The requirements of a literal are obtained by following the precondition-effect arcs in the causal graph defined by the rules, requirements$(l) = \{l' \vee requirements(l') \mid \exists \text{ arc}(l', l) \in CG_R\}$.

With a valid excuse φ, we can generate a subgoal G_φ to complete the task up to the point where φ is needed, as explained in the routine `SubgoalGeneration` (Algorithm 4). This algorithm starts by initializing the subgoal with the goal literals and finding the subgoal literals that have a dependency on the literals changed by the excuse in the causal graph. These subgoal literals are considered to be unreachable without the changes defined by the excuse, and thus they must be removed from the subgoal. Instead of only removing those literals, they are substituted by their dependencies in the causal graph. This process is repeated until all of the subgoal literals no longer depend on the excuse changes.

Figure 4.2 shows an example of the Cranfield benchmark domain (Sec. 2.2.2), where the robot has to mount several parts to complete the assembly. If the robot does not know how to place the front faceplate, the teacher would have to assemble all the other parts before demonstrating how to assemble the faceplate. To reduce the number of demonstrations required, the decision maker can create a subgoal where all of the known parts (the peg and the pendulum) should be mounted before requesting help.

The Cranfield Benchmark Setup

The REX-D algorithm proved to be very useful in the framework of the EU Project IntellAct[1]. In this project, the robot has to learn how to complete the Cranfield benchmark assembly (Sec. 2.2.2) with the help of a teacher. The robot starts with no previous knowledge and it learns different actions based on teacher demonstrations and experience. The decision maker that uses the REX-D algorithm is in charge of high-level reasoning, where it decides when to request teacher demonstrations and which actions to execute.

[1] IntellAct EU Project: http://intellact.sdu.dk/.

Figure 4.2: Intermediate state in the Cranfield benchmark scenario. Most of the assembly parts have not yet been placed in position.

Virtual Reality and Robot platform

As an initial step, all of the modules related to high-level reasoning were integrated into a virtual reality (VR) system with a simulated robot (Rossmann et al., 2013; Martínez et al., 2014a), which was very useful for testing the whole system before moving it onto the real robot platform because the same interfaces were used. The VR system provided the decision maker with symbolic states, the actions executed by the robot, and the information supplied to generate semantic event chains (Aksoy et al., 2011), which were used to recognize the actions executed by the teacher.

The VR setup comprised a multi-screen stereoscopic rear projection installation for showing the scenario and a wireless data glove with a tracking system for interacting with it. This system could keep track of hand movements, which were used to interact with objects in the scenario in the same manner as if they were in the real world. This VR system also simulated robot executions, including noise and action failures. The decision maker could request actions from this simulated robot in the same manner as it would request them from a real robot. Figure 4.3-left shows the teacher demonstrating an action in the VR system.

The same high-level reasoning skills were successfully integrated in a real robot platform (Savarimuthu et al., 2016, 2014). The robot could learn new actions from demonstrations, recognize the actions executed by the teacher, and provide REX-D with the skills to generate symbolic states that represented the scenario. Figure 4.3-right shows the setup used.

To demonstrate an action, the teacher provided the action name and its arguments, before moving the robot arm to perform the action. When a new unknown action was demonstrated,

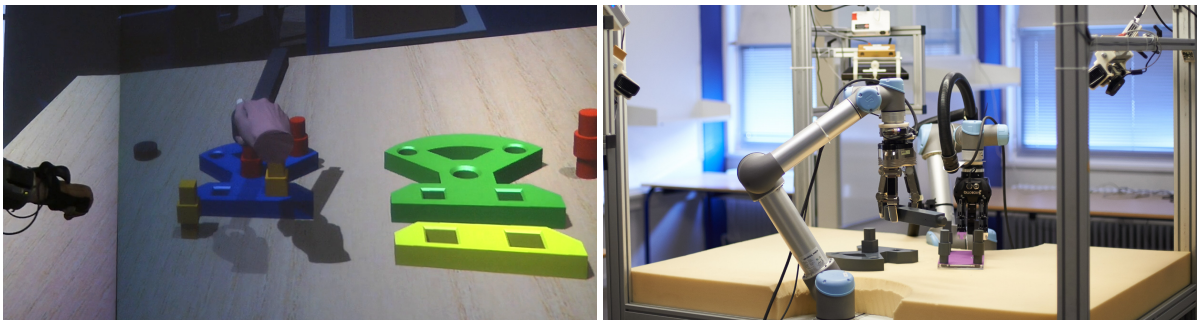


Figure 4.3: Execution of the Cranfield benchmark assembly. **Left:** A teacher moves the data glove, which is used to reproduce the movements of the simulated hand on the VR system. In this case, the teacher demonstrates how to place the pendulum in the assembly. **Right:** A robot arm places the pendulum using the perceptions obtained from the three RGB-D cameras.

programming by teleoperation (Savarimuthu et al., 2013) was used to learn the demonstrated action at a low level, thereby allowing it to be reproduced. The action was associated with the symbolic name given to the action. Transferring the knowledge to a high-level cognitive system could also be achieved with a symbolic representation that captured the interaction between objects and actions (Krüger et al., 2011; Randelli et al., 2013). However, in the present study, we assumed that the teacher supplied the name of the action that he executed for the robot.

The literals that represented the state used for high-level reasoning were obtained using the robot cameras. A 3D object recognition module (Buch et al., 2013) determined the different objects in the scene and their positions in the very first frame, which were then tracked as they moved (Papon et al., 2013), before transforming them into semantic event chains (Aksoy et al., 2011) to generate a symbolic state for planning and learning. Semantic event chains also facilitate the recognition of the actions executed by the teacher because they encode the changes in the relationships between objects at decisive time points during a manipulation procedure. Although the underlying perception algorithms work with continuous perceptions, REX-D only uses the information available before and after an action is executed because its cognitive processing is based on the state changes introduced by actions.

Although REX-D was finally integrated and tested in the real robot, the development process and experiments were mostly performed in the VR setup because it allowed us to conduct large numbers of replicates to obtain meaningful statistics, thereby facilitating the evaluation of the algorithm. The main limitations of using a real robot platform compared with a VR system are as follows,

- In a state, the objects need to be recognized perfectly, and thus they should be easily distinguishable.

- Objects need to be tracked continuously while the robot or the teacher interacts with the scenario. This process requires at least two cameras, which point in different angles to avoid occlusions, and the objects should not move faster than the tracker's frame rate.'

- The robot has to learn actions in a manner that allows them to be generalized to different object positions. In this study, these skills were limited to simple pick-and-place actions, which were sufficient to solve the tasks assigned.

Experimental Results

In this section, we present the results obtained using the REX-D approach. We compared the following configurations to assess the performance of the different algorithms explained in the present study:

- REX: The E^3 version of the REX algorithm (Lang et al., 2012). The results are reproduced from (Lang et al., 2012).

- REX-D basic: The REX-D algorithm (Sec. 4.3.1).

- REX-D rules: The REX-D algorithm with rule alternatives (Sec. 4.3.3). Only excuses that changed one literal were considered.

- REX-D rules+sub: The REX-D algorithm with rule alternatives and subgoals (Sec. 4.3.3).

In this section, we compare the REX and REX-D algorithms. Although the results are not directly comparable due to the addition of demonstrations, these experiments illustrate the improvements obtained by using a teacher that demonstrates a few actions.

Different scenarios were used to perform these comparisons. First, two problems from the IPPC 2008 (Sec. 2.2.1) were tested in order to make comparisons with other relational RL algorithms: *Exploding Blocksworld* and *Triangle Tireworld*. These are interesting problems with dead ends, which make extensive use of relational actions.

The robot setup for the REX-D algorithm was devised using the Cranfield benchmark (2.2.2). To demonstrate the performance of REX-D in this scenario, extensive experiments were conducted using this setup in a VR system. The following features of our approach were assessed.

- The capacity to cope with stochastic domains compared with deterministic domains.

- The generalization capabilities over similar tasks.

- The improvements obtained using rule alternatives in terms of reducing the number of demonstration requests needed.

- The improvements obtained by using subgoals in terms of reducing the number of demonstration requests.

To evaluate the relational algorithms we set the threshold ζ to consider known states and actions. Similar to other RL algorithms (Lang et al., 2012; Li, 2009), we set ζ heuristically because it cannot be derived from theory when using a heuristic learner. To obtain results that are comparable to those produced by REX, we used a value employed in previous studies ($\zeta = 2$), which obtains good results in many scenarios. In addition, the decision maker was limited to executing 100 actions per episode. After these actions were executed, the episode was marked as failed and the experiment continued with the following episode. All of the REX-D configurations used Gourmand (Kolobov et al., 2012b), which is probabilistic planner; and Pasula et al. (2007)'s learner, which is able to learn probabilistic relational domains.

The quantitative results presented in this section were only obtained in the VR system since a very large number of teacher requests were required to obtain statistically significant results, which would have been tedious for a human teacher. Therefore, an automated teacher that planned with the correct rule set was used to perform the experiments described in this section. This automated teacher also considered the guidance provided by the system when generating plans. Because it obtained optimal policies, the results were the same as those that would have been obtained with a human teacher.

Experiment 1: Comparing REX and REX-D in Exploding Blocksworld (IPPC)

The results obtained in the *Exploding Blocksworld* domain (Sec. 2.2.1) are shown in Fig. 4.4. The performance of the REX-D algorithm was clearly better because demonstrations allowed it to quickly learn the correct usage of the different actions. By contrast, REX had to execute an average of 120 extra exploratory actions to learn how to apply the actions correctly. Given that each action may take several seconds to complete, the amount of time saved can be very important.

The main problem for algorithms without demonstrations is finding positive examples. As explained in Proposition 4.1, we can calculate the number of experiences that would be required to obtain a positive example with the *pick-up(X,Y)* action in the worst case. For simplicity, we assume that this action does not have deictic references. The representation of this domain with NDR rules comprised four actions represented by six rules, and eight different types of literals. As the number of literals $|L_r| = 16$, REX-D would require $O(2^{16})$ different experiences in the worst case. Nevertheless, having one positive example, such as *pick-up(box5, box3)*, allows its preconditions to be learned with just 16 different experiences, i.e., one to test each of the 16 literals in the state that may be related to the action, as explained in Proposition 4.2. Although

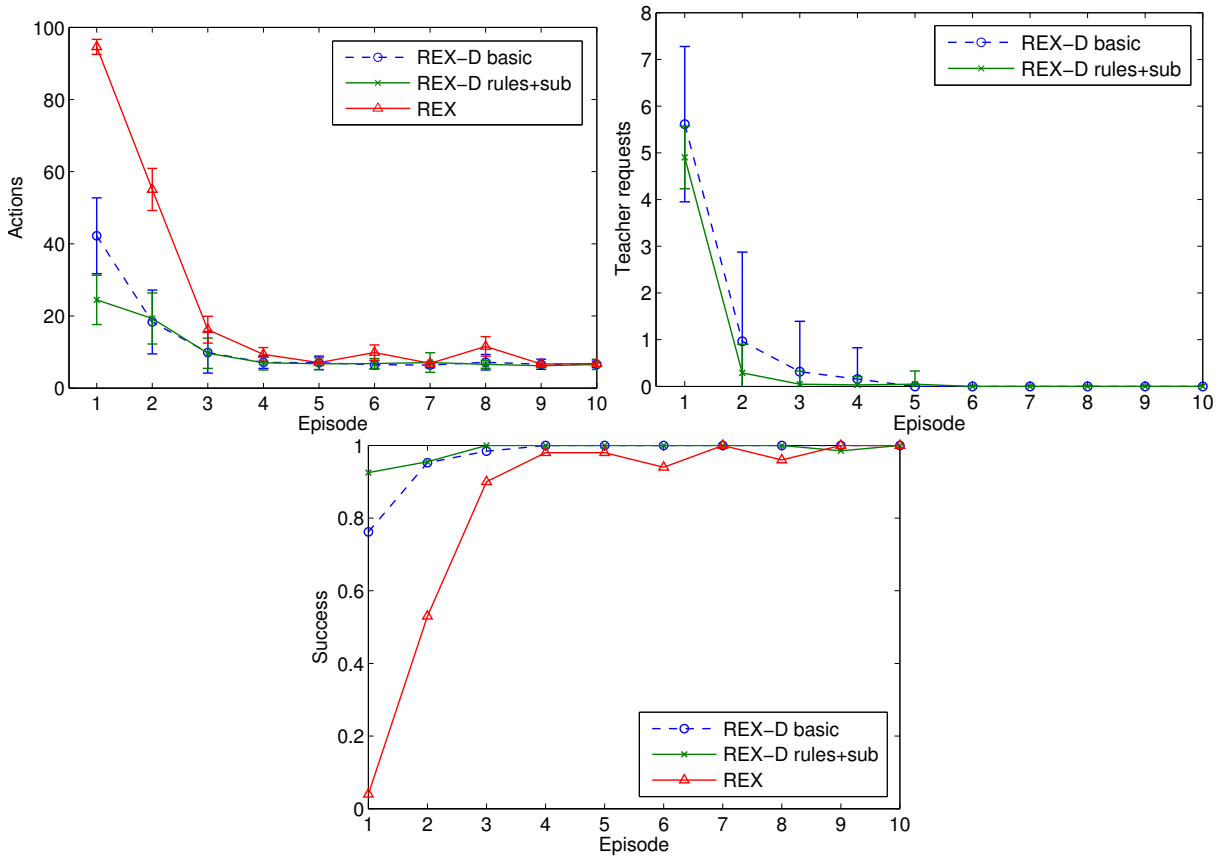


Figure 4.4: Experiment 1 (Exploding Blocksworld, problem 5): The robot started with no knowledge of the actions in the first episode. The results shown are the mean and standard deviations obtained over 50 runs. REX data were obtained from (Lang et al., 2012). **Upper left:** the number of actions executed in each episode. **Upper right:** the number of teacher demonstrations requested. **Bottom:** the success ratio of the algorithms.

the actual number of experiences required to learn this domain is much lower (the rule only has four preconditions and the initial state also limits the number of incorrect experiences that can be performed), these theoretical numbers reflect the reduction in complexity obtained through demonstrations.

Using rule alternatives and subgoals reduced the number of demonstrations required. However, the alternative paths that they generated sometimes led the robot to a dead end, whereas demonstrations took the optimal path.

The number of teacher interactions varied between five and seven, which was sufficiently low considering the improvements in the overall number of actions required and the improvements in the success ratio. It should also be noted that the teacher was no longer required after the fifth episode.

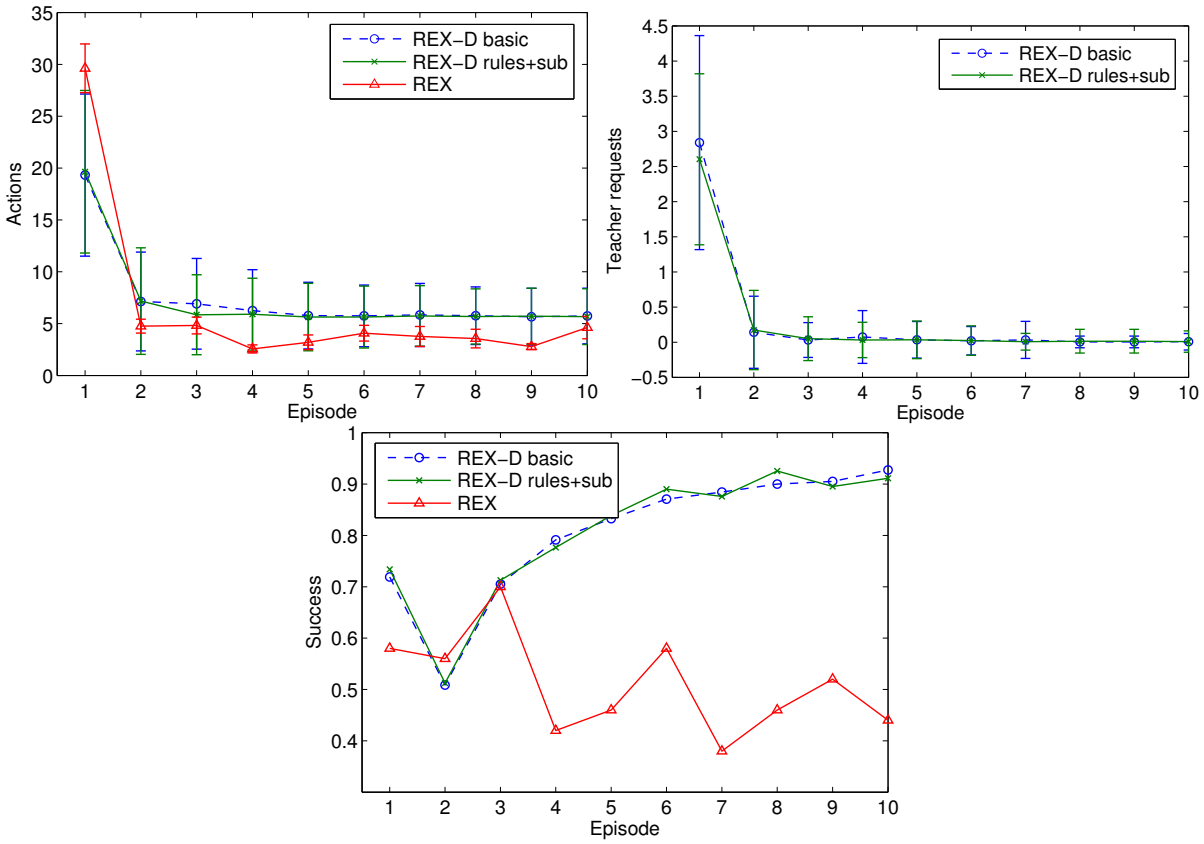


Figure 4.5: Experiment 2 (Triangle Tireworld, problem 1): The robot started with no knowledge of the actions in the first episode. The results shown are the means and standard deviations obtained from 100 runs. REX data was obtained from (Lang et al., 2012). **Top left:** the number of actions executed in each episode. **Top right:** the number of teacher demonstrations requested. **Bottom:** the success ratio of the algorithms.

Experiment 2: Comparing REX and REX-D in Triangle Tireworld (IPPC)

The results obtained in the *Triangle Tireworld* domain (Sec. 2.2.1) are shown in Fig. 4.5. The difficulty of this domain was the dead ends that the agent encountered when it had a flat tire and no spare tires were available. Both algorithms failed to learn how to change a tire if no flat tires were obtained in a location with a spare tire during the initial episodes. After they considered the actions as known, they always took the shortest dangerous path because they did not know that tires could be changed. The main problem for REX is that it moved through random paths initially and it could become confident about its model before learning how to fix a tire. REX-D exhibited better behavior because the initial demonstrations directly moved the car through roads with spare tires and a flat tire could be replaced if the car had a spare tire in its location. REX-D required more actions per episode because it selected longer and safer paths.

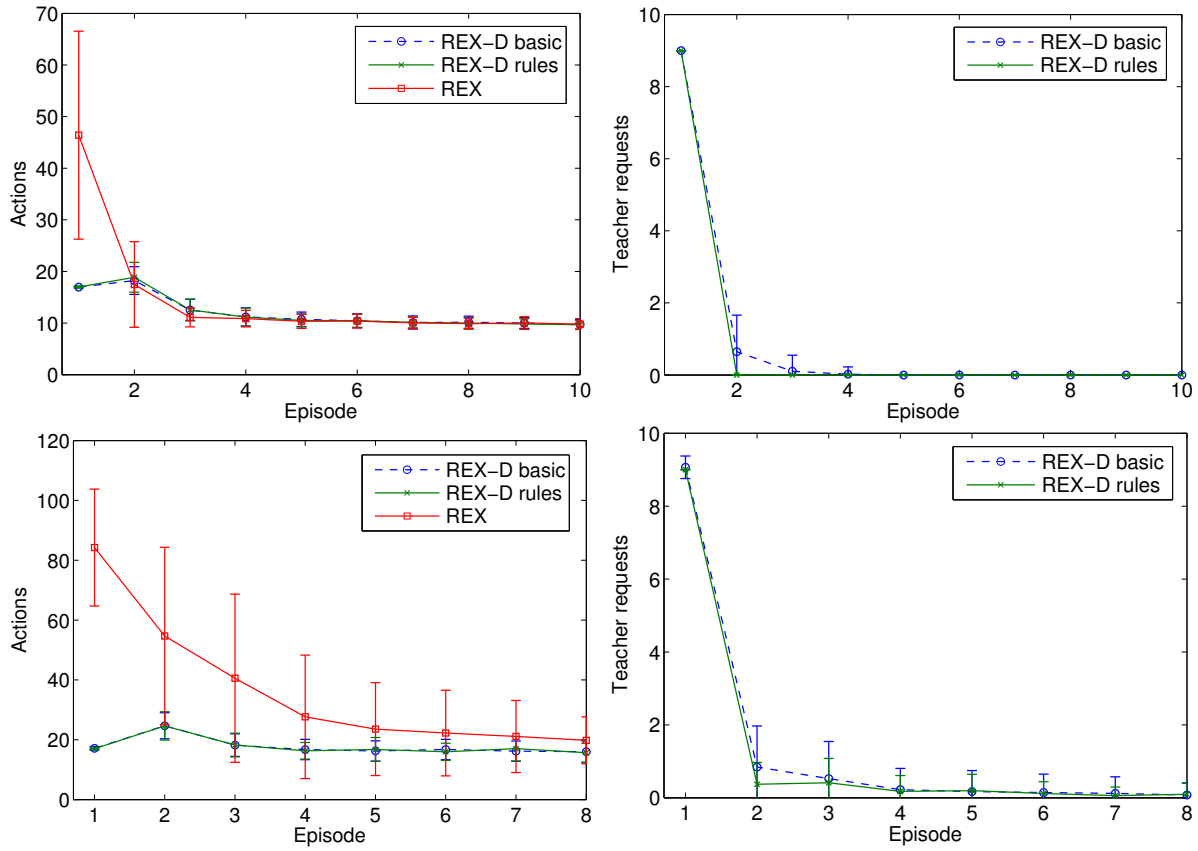


Figure 4.6: Experiment 3 (Standard Cranfield benchmark): The robot started with no knowledge of the actions in the first episode. The standard Cranfield benchmark assembly had to be completed in each episode. As there were no dead-ends and a teacher was available, the success ratio was 1. The results show the mean and standard deviations obtained over 100 runs. **Top:** deterministic Cranfield benchmark. **Bottom:** stochastic Cranfield benchmark. In this benchmark, the actions had a success ratio of 60%. **Left:** the number of actions executed in each episode is shown. **Right:** the number of teacher demonstration requests is shown.

Because it was easy to learn the preconditions in this domain and the difficulty was avoiding dead ends, rule alternatives and subgoals did not improve the results.

Experiment 3: Standard Cranfield benchmark

We tested how well REX-D performs in stochastic domains compared with deterministic domains. These experiments were performed using the standard Cranfield benchmark in the VR system, where we changed the success probability of the actions to 60% for the stochastic case. There were no dead ends and the only change was the success ratio of the actions; thus, the differences in the results simply indicated the changes obtained with noise when learning.

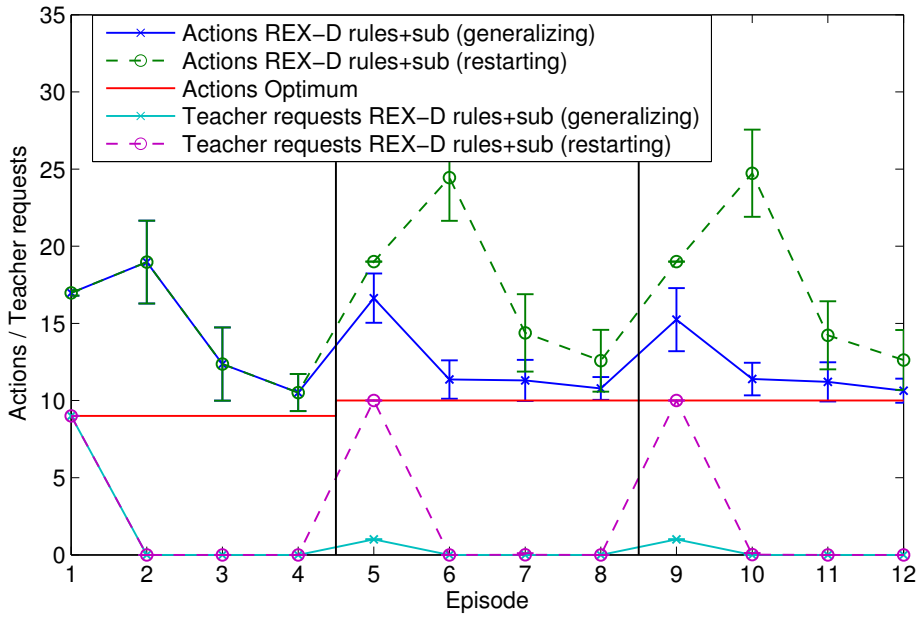


Figure 4.7: Experiment 4 (Generalizing tasks in the Cranfield benchmark): Three different Cranfield benchmark problems had to be completed, which are separated by vertical black lines. The optimal number of actions required to complete the task changed because the variants of the domain used demanded that an additional action was completed. Because there were no dead ends and a teacher was available in case of failure, the success ratio was 1. The results represent the means and standard deviations obtained from 50 runs. The number of actions performed and the number of demonstration requests are shown. **REX-D (generalizing)** started with no previous knowledge of the actions in the first episode, but it maintained its experience when the problem changed. **REX-D (restarting)** started with no previous action knowledge each time the problem changed.

Figure 4.6 shows the results obtained. The REX-D algorithm produced good results in both the deterministic and stochastic domains. By contrast, REX had to explore intensively, especially in the stochastic case. Using REX-D with rule alternatives significantly reduced the number of demonstrations requested. In the deterministic case, only the initial demonstrations were required, while only a few requests were made in the probabilistic case.

Experiment 4: Generalizing tasks

In this experiment, the decision maker had to complete different tasks. The changes in the task required new actions to be learned to solve problems that had not been presented previously. Three different variants of the Cranfield benchmark were used.

- Episodes 1 - 4: The standard Cranfield benchmark was used.

- Episodes 5 - 8: The task started with a peg in a horizontal position. The action to place pegs failed if they were not in a vertical position, and thus, the peg had to be repositioned by learning a new action.

- Episodes 9 - 12: Initially, the separator was placed without any pegs. To complete the scenario successfully, the robot needed to place the pegs but they could only be placed when the separator was not in place, so it had to be removed first, which required that a new action was demonstrated.

Figure 4.7 shows the results obtained with the REX-D algorithm. Each time the task was changed, after executing a few actions, the system realized that it needed a new action and requested a demonstration from the teacher. As more episodes were added, although some changes were made to the task, the results became closer to the optimum. By contrast, when REX-D restarted after a problem change, numerous demonstrations and exploratory actions had to be executed to complete the task because everything had to be learned again.

Experiment 5: Goal changes

This experiment demonstrated the advantages of using subgoals. The decision maker started learning from scratch with the standard Cranfield benchmark assembly and a change was made in the scenario after four episodes, where the goal changed such that a new special front faceplate with no requirement for a pendulum had to be placed.

The REX-D algorithm without subgoals requested a demonstration from the teacher before placing any parts because it did not know how to place the new front faceplate, which was required to reach the goal. This new part required all the pegs and the separator to be placed in advance, so the teacher had to position them before teaching the new action. However, the use of subgoals allowed the robot to complete the known parts of the assembly, i.e., the pegs and the separator, before issuing a help request to the teacher to learn only the missing action.

Figure 4.8 shows the improvement obtained after adding subgoals. The REX-D algorithm with subgoals only requested one demonstration from the teacher, whereas the standard REX-D required that all the actions were executed.

Dangerous actions

In this section we propose a new method that extends REX-D (Sec. 4.3) to avoid exploring dangerous parts of the state space as well as to safely refine such parts of the model, so that the planner can avoid falling in dead-ends. The following shortcomings are overcome with dead-end avoidance:

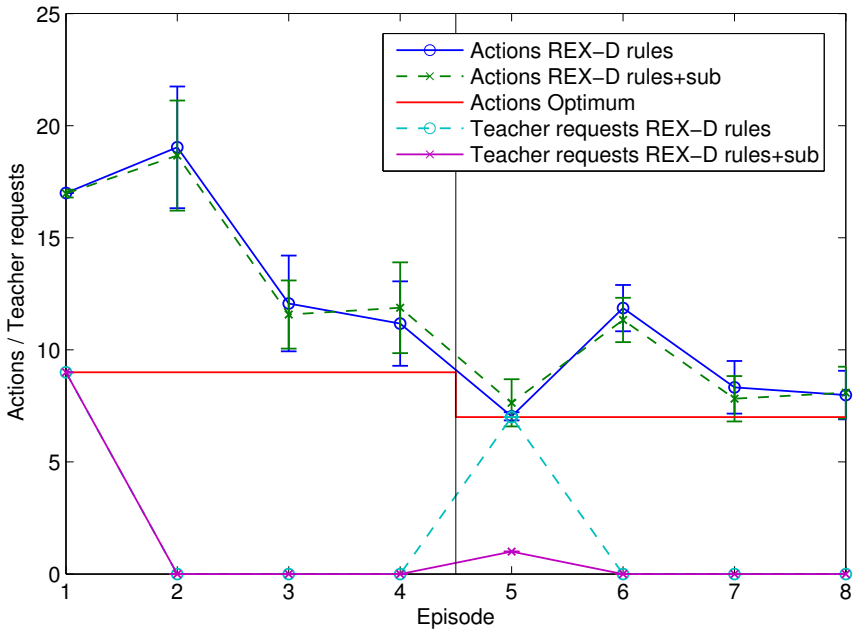


Figure 4.8: Experiment 5 (Goal changes): The standard Cranfield benchmark assembly is tackled during the first 4 episodes, and a special front faceplate that does not require a pendulum has to be placed during the last 4 episodes. The robot starts with no previous knowledge about the actions in the first episode. As there are no dead-ends and a teacher is available, the success ratio is 1. The results shown are the mean and standard deviations obtained over 50 runs. Both the amount of actions performed and the number of demonstration requests are shown.

- Exploration of dead-ends: until all the states that lead to a unrecoverable error get explored, a RL algorithm would fall into that dead-end repeatedly.

- Suboptimal policies: Introducing a heuristic model learner, relational generalizations and teacher demonstrations has the advantage of reducing exploration (Sec. 4.3), but it also has drawbacks. Using a relational count function implies that not all states are explored before considering them as known, since all states within a context are assumed to behave likewise. Therefore, state-action pairs that could be needed to attain the best policy might not be visited, and thus their contexts would not be learned. This lack of exploration may lead to models that yield suboptimal policies that fall more frequently into dead-ends.

First, we explain how the dangerous literals that model unrecoverable errors are detected, and how these dangerous literals are used to find dangerous rules. Then we describe how the decision maker analyzes plans to check whether they have a high-risk of reaching a dead-end. Finally the procedures to avoid dead-ends are presented.

Detecting Dangerous Literals

The first requirement to avoid unrecoverable errors is the ability to detect their causes. From a planning point of view, the dangerous literals are those that when present in the state, the planner can no longer find a sequence of actions that reaches the goal. Therefore dangerous literals represent unrecoverable errors. We use planning excuses (Sec. 2.1.7) to identify dangerous literals. If an excuse is obtained in a state with an unrecoverable error, the excuse changes will show the dangerous literals that are making the task unsolvable. Thus the list of dangerous literals D_φ can be extracted as $D_\varphi = s \triangle s_\varphi \mid \langle C_\varphi, s_\varphi \rangle = \varphi$, where s is the current state and $\triangle$ denotes the symmetric set difference.

Library of Dangerous Literals

The decision maker maintains a library of dangerous literals L_d. This library contains a list of pairs $\langle d_l, d_r \rangle$ where d_l is the literal that has caused an unrecoverable failure and d_r the acceptable risk for that literal. The acceptable risks are always initialized to 0. This library is updated or used in the following cases:

- Whenever a dead-end is reached: an excuse φ is obtained, and for each literal l changed by the excuse $l \in D_\varphi$, we add the pair $\langle l, 0 \rangle$ to the library if l had not been added before $\nexists l \mid \langle l, d_r \rangle \in L_d$.
- Whenever the teacher confirms that a dangerous rule has an acceptable risk: the risk associated to the literals in the rule effects are updated (Sec. 4.4.2).
- Whenever the model T is updated (e.g. by the learner): any rule $r \in T$ whose effects include a literal $l \in \mathit{eff}(r, i) \mid \langle l, d_r \rangle \in L_d$ is marked as dangerous.

Avoiding Dead-ends

To avoid dead-ends, the decision maker has to skip exploration of potentially dangerous rules and request help from the teacher when the learned model leads to dead-ends.

Safe Exploration

To explore an action, rules have to be considered known (using Eq. 4.1) and safe. To check the safety, for each $\langle d_l, d_r \rangle \in L_d$ the decision maker checks that the risk for each dangerous literal is low enough. A rule r can be explored if the sum of the probabilities of getting the dangerous literal $p(d_l \mid r) = \sum_i p(r, i) \, \forall \{ \langle \mathit{eff}(r, i), p(r, i) \rangle \in r \mid d_l \in \mathit{eff}(r, i) \}$ is lower than the acceptable risk d_r for that literal d_l.

Algorithm 5 Avoid dangerous rules

Input: Planned dangerous rule r, current state s, library of dangerous literals L_d

1: $L'_d = L_d$ ▷ *Copy the library to store expected dead-end risk from* s
2: $d'_r = 0 \ \forall \langle d'_l, d'_r \rangle \in L'_d$ ▷ *Initialize expected risks to 0*
3: **for** each effect $\langle \mathit{eff}(r,i), p(r,i) \rangle$ in rule r **do**
4: $s' = s$
5: Add literals in $\mathit{eff}(r,i)$ to s'
6: Plan(s')
7: **if** dead-end **then**
8: **for** each dangerous literal $d_l \in \mathit{eff}(r,i)$ **do**
9: Get $d'_r \mid \langle d'_l, d'_r \rangle \in L'_d,\ d_l = d'_l$
10: Update $d'_r = d'_r + p(r,i)$
11: **if** $d'_r > d_r \mid \langle d_l, d_r \rangle \in L_d$ **then**
12: $danger = true$
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: **if** $danger$ **then**
18: Plan without rule r
19: **if** Obtained new plan **then**
20: Analyze new plan
21: **else**
22: Request teacher confirmation
23: **if** confirmed as safe **then**
24: Update acceptable risks in L_d
25: **else**
26: Get demonstration from the teacher
27: **end if**
28: **end if**
29: **end if**

Learning Safe Models

Whenever a new action is planned, the decision maker checks if the state-action pair corresponds to a dangerous rule. In this case a special procedure is executed to analyze the planned action safety.

Algorithm 5 summarizes the procedure to detect cases where there is danger of falling into a dead-end. First, in lines 4-11, the possible effects $\mathit{eff}(r,i)$ of the planned dangerous rule r are simulated, and the planner is used to check whether any of them may lead to an dead-end. After this analysis, the decision maker knows the expected dead-end probability d'_r associated to each dangerous literal d'_l. In lines 12-14, if the expected dead-end probability is greater than

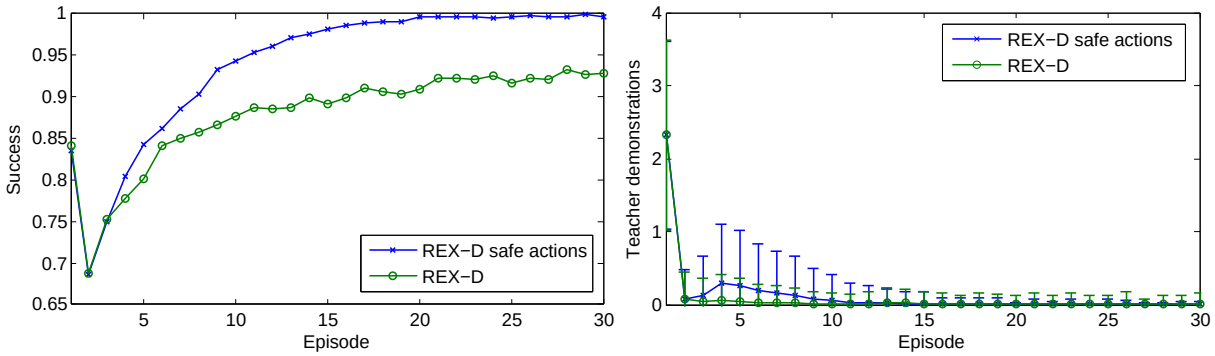


Figure 4.9: Experiment 1 (Triangle Tireworld, problem 1). The results shown are the means and standard deviations obtained from 2500 runs. The exploration threshold is $\zeta = 2$. **Left:** the success ratio of the algorithms. **Right:** the number of teacher demonstrations requested.

the acceptable risk for any literal $d'_r > d_r \; \forall \{d_l, d_r, d'_r \mid \langle d_l, d_r \rangle \in L_d, \langle d_l, d'_r \rangle \in L'_d\}$, either an alternative plan is found or teacher help is required.

After finding a risky rule, the decision maker plans again using a model that does not contain the dangerous rule r (lines 18-21). If a safe plan is obtained, it can be executed, and otherwise it is also analyzed using Algorithm 5. However, if a plan cannot be found, the robot interacts with the teacher to confirm the safety of the originally planned action (lines 22-24). The confirmation consists in notifying the teacher about the action that the robot intends to execute, and the teacher can either confirm that it is safe or demonstrate a different and safer action instead. To handle domains where dead-ends are not completely avoidable, if the teacher confirms a rule as safe, the risks associated to the dangerous literals in that rule are updated $d_r = max(d_r, d'_r + \epsilon)$, considering that there was an estimated d'_r risk of dead-end for that literal. However, if the rule is actually dangerous (lines 25-27), a safer action will be presumably demonstrated by the teacher. This new action will be learned and added to the model, so that the planner will have the option to choose it afterwards.

Experimental Results

Two experiments were carried out to validate our proposal. The first one was a simulated problem from the Triangle Tireworld domain of the IPPC 2008 (Sec. 2.2.1). The second was a real robotic task, which consisted in clearing the tableware on a table (Sec. 2.2.2).

Experiment 1: Triangle Tireworld

The Triangle Tireworld (Sec. 2.2.1) is a challenging domain where RL approaches with a low exploration threshold ζ can fall easily into recurrent dead-ends. If the robot does not get a flat

tire while it explores locations where spare tires are available, it will not learn how to change a tire and therefore it will always choose the shortest path to the goal where no spare tires are available.

However, the dead-end avoidance permits learning correctly this problem without requiring a large exploration threshold and thus not increasing the number of exploratory actions needed to learn the domain. After the robot realizes that moving may lead to flat tires, it will confirm the moving actions with the teacher, who will recommend the safer paths with spare tires instead of the shorter ones. Once the car gets a flat tire with a spare tire available, it will learn how to change tires and perform successfully afterwards. Figure 4.9 shows the advantages of recognizing and dealing with dangerous rules, where a few extra teacher demonstration requests allow the robot to have a much better success ratio, getting a 98% success ratio after 15 episodes, compared with 88% that would have been obtained without considering dangerous rules (Fig. 4.9 Left). The cost of this improvement is very low, as just one extra teacher demonstration is needed in average (Fig. 4.9 Right).

Experiment 2: Tableware Clearing

This experiment consists in clearing the tableware on a table (Sec. 2.2.2). The robot has to take plates, cups and spoons from a table to the kitchen. However, moving to the kitchen is a costly action, and therefore the robot has to stack the tableware before taking them, minimizing the time spent.

The actions *putPlateOn*, *putCupOn* and *putForkOn* place the corresponding object on another one. If a plate is placed on a cup or spoon, or a cup is placed on a spoon, it may fall and break. (For practical reasons, we used plastic tableware in the experiments so that they wouldn't actually break. Instead, they were considered to be broken after falling from a considerable height.) There are another 3 inverse actions to place each of the objects back on the table, and one action *moveStacks* to take the stacks to the kitchen.

The reward is based on the time spent, but there is also a high penalty for each broken object. Each action reduces the reward by 0.05, while the *moveStacks* action reduces it by 1 for every stack, and by 5 for every broken object. Finally, when the table is cleared, a reward of 5 is obtained.

The robotic setup consists of a WAM arm equiped with a gripper, and a RGB-D camera that is positioned on the ceiling. To generate symbolic state representations for the decision maker, the perception system recognizes the tableware on the table and their relative positions, and maintains a believe state that is needed to tackle the occlusions when an object is placed on top of another. The movement primitives to execute pick and place actions are also available in the robot, and the interaction with the teacher is implemented using a PC located near the robot.

Figure 4.10: Tableware clearing setup. **Left:** Example of the initial setup. **Middle:** Example of a perfectly stacked pile of tableware. **Right:** Example of a badly stacked pile, which if moved will result in objects falling from the pile.

REX-D	Success 1 pile (reward)	Success 1 or 2 piles (reward)	Broken object
Dead-end avoidance	67% (3.78 ± 0.02)	87% (3.54 ± 0.43)	13%
Standard REX-D	60% (3.58 ± 0.13)	60% (3.58 ± 0.13)	40%

Table 4.1: Experiment 2 (Clearing Tableware). The results shown are the success ratios, and the means and standard deviations of the accumulated rewards obtained from 15 runs. The scenario was initialized with 2 plates, 2 cups, and 1 spoon. The exploration threshold is $\zeta = 3$. **Success 1 pile:** Experiments where the robot successfully (i.e. with no broken objects) piled everything into one pile (and the accumulated reward obtained in the successful executions). **Success 1 or 2 piles:** The same as before, but obtaining 1 or 2 piles. **All:** Accumulated reward counting all the executions. **Broken objects:** Experiments where the robot broke at least one object.

In this experiment two different models were learned using the REX-D algorithm (Sec. 4.3): one was learned with the dead-end avoidance method presented in this section, and the other with the standard REX-D algorithm. Both were learned during 8 episodes in which the teacher was available to answer the robot inquiries. The initial setup consisted of 2 plates, 2 cups, and 1 spoon as shown in Fig. 4.10.

The two models were then used to complete the tableware clearing task 15 times, and the results are shown in Fig. 4.1. The model with dead-end avoidance reduced the number of executions with broken objects drastically, from 40% to only 13%. Note that robot actions and perceptions are not perfect, and reducing more the number of broken objects would be very complicated without improving these actions.

Moreover, the model with dead-end avoidance did not always stack everything into a single pile. The reason is that the risk of breaking an object (and thus falling into a dead-end) was not worth stacking the two piles together. Therefore the decision maker decided to finish with two piles in 20% of the runs.

V-MIN

In this section we present the V-MIN algorithm, that extends the REX-D algorithm (Sec. 4.3) to provide a more general and complete approach.

Exploration in V-MIN is done with the R-MAX (Brafman and Tennenholtz, 2003) algorithm and the relational generalizations in REX (Lang et al., 2012). The R-MAX algorithm plans in a model where unknown state-action pairs yield the maximum reward R_{max} to encourage active exploration, and the REX algorithm applies the context-based density function in Eq. 4.1 to improve the generalization over different states. As commented in Sec. 4.3, these generalizations assume that all states within a context behave likewise, and thus not all state-action pairs in a context are visited before considering the context as known. Therefore, important state-action pairs may not be visited, and thus their contexts (i.e. rules) not learned.

Similar to REX-D (Sec. 4.3), V-MIN uses teacher demonstrations to learn new actions and contexts, and improve its performance. Nevertheless, the integration of demonstration requests is different: V-MIN uses the concept of V_{min}, which is the minimum expected value $V^{\pi}(s)$. If the planner cannot obtain a $V^{\pi}(s) \geq V_{min}$, it actively requests help from a teacher, who will demonstrate the best action $a = \pi^{*}(s_i)$ for the current state s_i. Therefore, when exploitation and exploration can no longer yield the desired results, demonstrations are requested to learn yet unknown actions, or to obtain demonstrations of actions whose effects were unexpected (i.e. new unknown contexts). Note that in the special case where a value $V(s) \geq V_{min}$ cannot be obtained anymore, which is a dead-end, the teacher signals that the episode has failed instead of demonstrating an action.

To include V_{min} in the model, an action "TeacherRequest()" is defined which leads to an absorbing state with a V_{min} value. The "TeacherRequest()" action asks for a demonstration from the teacher.

V-MIN is a more general algorithm than REX-D as it can work in any domain in which the value function has to be optimized. In contrast, REX-D is limited to goal-driven problems, and it may settle with suboptimal solutions as long as they can reach the goal. Moreover, V-MIN has the following features:

- A high V_{min} forces the system to learn good policies at the cost of a higher number of demonstrations and exploratory actions, whereas a lower V_{min} leads to a faster and easier learning process, but worse policies are learned.

- The teacher can change V_{min} online until the system performs as desired, forcing it to find better policies.

- V-MIN can adapt well to new tasks. Context generalizations and teacher demonstrations

Algorithm 6 V-MIN

Input: Reward function $\mathcal{R}$, confidence threshold ζ **Updates:** Set of experiences E

```
Observe state s_0
loop
    Update transition model T according to E
    Create T_vmin(T, R, ζ)
    Plan an action a_t using T_vmin
    if a_t ==“TeacherRequest()” then
        a_t = Request demonstration
    else
        Execute a_t
    end if
    Observe new state s_{t+1}
    Add {(s_t, a_t, s_{t+1})} to E
end loop
```

are specially useful, as they permit transferring a lot of knowledge and learn new actions and contexts as required. Moreover, the V_{min} value can be modified to adapt to changes in the tasks. Using Taylor and Stone (2009)'s transfer learning categorization, the new task can change the start state, the goal state, and the number of objects without requiring further learning. New actions can also be incorporated through demonstrations, and the reward function can change without having to modify the rest of the model.

The expressibility of the models used by V-MIN depends on the planner and learner being used. In this thesis we use model learners and planners that can tackle uncertainty in the actions but not in the observations. To tackle partially observable domains we would just have to integrate a learner and a planner that could handle them, without needing to perform any further changes in the V-MIN algorithm. As examples, we have successfully integrated V-MIN with the learner proposed by Pasula et al. (2007), the learner presented in Chapter 5, and the planners Gourmand (Kolobov et al., 2012b), PROST (Keller and Eyerich, 2012) and PRADA (Lang and Toussaint, 2010).

Algorithm

In this section we assume that the model used consists of NDR rules (Sec. 2.1.1). However, the same would apply to the planning operators explained in Sec. 2.1.1.

Algorithm 6 shows a pseudocode for V-MIN. This algorithm maintains a model T that represents the robot actions as described in Sec. 2.1.1. This model is updated every iteration to adapt to newer experiences using a model learner (Sec. 2.1.4). The T_{vmin} model is then created to plan the action to execute, implicitly selecting exploratory actions until all the relevant contexts

Algorithm 7 Create T_{vmin}

Input: Model T, reward function $\mathcal{R}$, confidence threshold ζ
Output: V-MIN model T_{vmin}

1: $T_{vmin} = T$
2: **for all** $r \in T_{vmin}$ **do**
3: **if** $known(r, \zeta)$ **then**
4: $\mathcal{R}(r) = R_{max}$
5: **end if**
6: $pre(r) = pre(r) \wedge -$*finished*()
7: **for all** *eff*$(r, i) \in r$ **do**
8: *eff*$(r, i) =$ *eff*$(r, i) \wedge$ *started*()
9: **end for**
10: **end for**
11: Add rule $r_{vmin} =$"TeacherRequest()" to T_{vmin}
12: $\mathcal{R}(r_{vmin}) = V_{min}$

are considered known, and requesting teacher demonstrations whenever a value $V^{\pi}(s) > V_{min}$ cannot be obtained.

Algorithm 7 shows how to create T_{vmin} as an extension of T that includes implicit exploration and teacher demonstration requests. First, to check whether a rule is known, equation 4.1 is used to count if the rule covers at least ζ experiences (note that equation 4.1 includes relational generalizations). As in the R-MAX algorithm, every unknown rule is given the maximum reward R_{max} to explore implicitly following the principle of "optimism under uncertainty". Moreover, to include teacher demonstration requests, the "*-finished()*" literal is added to the preconditions of all rules, the "*started()*" literal to all rule effects, and finally, the special rule "TeacherRequest()" is added with reward V_{min}.

V-MIN requires that plans containing the special action "TeacherRequest()" must have a value of V_{min}. If "TeacherRequest()" were combined with other actions in the same plan, their combined value would be $V_{min} + V(s')$, and thus we are limiting "TeacherRequest()" to be the only action in a plan when selected. This behavior is obtained through the use of the two new literals: *started* and *finished*. In sum, the following changes are made to T_{vmin}:

- Create a rule for the "TeacherRequest()" action with $pre(teacher) = -$*started*() and a deterministic effect *eff*$(teacher, 1) =$ *finished*().
- Change all preconditions $pre(r) = pre(r) \wedge -$*finished*() $\forall r \in T$.
- Change all rule effects *eff*$(r, i) =$ *eff*$(r, i) \wedge$ *started*() $\forall i, r \in T$.
- Change the state $s = s \wedge -$*started*() $\wedge -$*finished*().

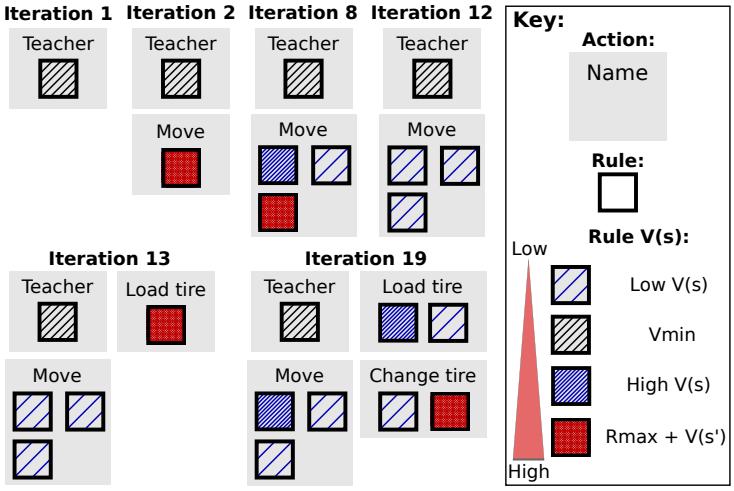


Figure 4.11: Example of the model generated by V-MIN in the *Triangle Tireworld* domain (Sec. 4.5.4). **Iteration 1:** the only action is requesting a demonstration. **Iteration 2:** a rule to model the new demonstrated action "Move" is available. It has a R_{max} reward as it hasn't been explored yet. **Iteration 8:** After several exploratory actions, "Move" is modelled with 3 rules, 2 of which are already considered to be known. **Iteration 12-13:** a high value plan can't be found, so a new action is demonstrated. **Iteration 19:** new actions have been demonstrated and explored. Two rules that lead to high $V(s)$ are available. Note that although rules with R_{max} reward exist, they may not be reachable from the current state. For example, one "Change tire" rule requires a flat tire.

"TeacherRequest()" requires the precondition "-started()", and all other rules will have the "started()" literal as effect, so "TeacherRequest()" can only be selected as the first action. Moreover, all rules require the "-finished()" precondition, and "TeacherRequest()" has "finished()" as effect, so it can only be selected as the last action. Considering that "TeacherRequest()" can only be the first and last action, whenever selected it will be the only action in the plan.

Finally, Fig. 4.11 shows an example of how T_{vmin} evolves as actions get demonstrated and executed.

Reward Function

The V-MIN algorithm requires the reward function to be known. Global rewards have to be defined explicitly in terms of literals and the number of actions executed. Actions can also provide rewards which have to be either specified by the teacher during the demonstration, or perceived by the system, as V-MIN does not estimate rewards.

The value of V_{min} can be defined as a constant or as a function, and has to be updated as the system completes the task. When a discount factor γ exists, this discount has to be applied to the V_{min} value as the goal is approached.

In real-world applications, the teacher selects the V_{min} value. Although rewards can be defined with a measure that can be understood by the teacher (e.g. the time spent or the number of objects to be manipulated), selecting appropriate values for V_{min} can be complicated. As V_{min} can be modified online, the teacher can increase it until the system behaves as desired. This allows a teacher to use V-MIN without requiring in-depth knowledge about its reward function.

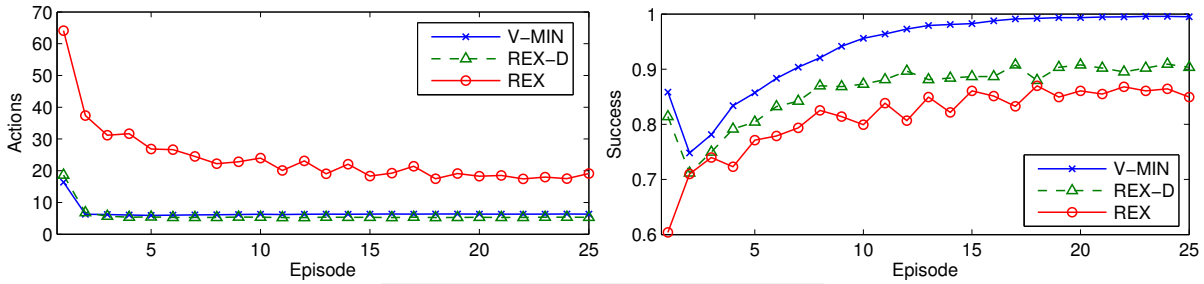
Performance Analysis

R-MAX is PAC-MDP (Strehl et al., 2009), which means that it learns near-optimal behavior in MDPs with a polynomial number of samples $\widetilde{O}(|S|^2|A|/(\epsilon^3(1-\gamma)^6))$. The generalization to relational domains has also been proven to be PAC-MDP under the assumption that a KWIK learner and a near-optimal planner are used (Lang et al., 2012). As the required set of actions is demonstrated during the learning process, and the exploration method is the same, the upper bound for the sample complexity of R-MAX holds also for V-MIN to get a policy π_{vmin} such that $V^{\pi_{vmin}}(s) \geq V_{min}$.

Nevertheless, the V-MIN algorithm performs much better in average situations. The use of generalizations and teacher demonstrations provides various structural assumptions that lead to an improved performance. The $|S|^2|A|$ product represents the $|S||A|$ state-action pairs that can be executed, which can lead to $|S|$ different states. Applying V-MIN to standard domains reduces this $|S|^2|A|$ number greatly:

- Standard domains usually have a small number of effects for each state-action pair, and furthermore, NDR rules permit modelling very rare effects as noise, reducing a $|S|$ factor to a small constant K_{eff} that is the maximum number of effects that a rule may have in the domain.

- Learning the $|S||A|$ state-action pairs can be reduced to a complexity proportional to $\log_2(|S|)|R|$. Applying equation 4.1, the exploration is carried out over $|R|$ state-action contexts (where R are the rules needed to model the system). However, rule contexts are not provided, and finding them is proportional to $|S||A|$ as the system does not know which state-action pairs describe them (Walsh, 2010). Using the V-MIN algorithm, the teacher demonstrations provide positive examples of the relevant contexts. As the preconditions of a rule (which define the contexts) are a conjunction of literals, only literals in the state s_d where the demonstration was executed may be in the rule preconditions. Thus, testing every one of these literals to see if they are really a part of the preconditions results in requiring $|s_d| = \log_2(|S|)$ samples for each context.



	V-MIN	REX-D
Teacher	3.87 ± 1.53	2.65 ± 1.19

Figure 4.12: Experiment 1 (Triangle Tireworld, problem 1): Means over 250 runs. **Top:** the number of actions executed in each episode. **Middle:** the success ratio. **Bottom:** the total number of teacher demonstrations requested.

In conclusion, the expected sample complexity of the V-MIN algorithm is proportional to $K_{\mathit{eff}} \log_2(|S|)|R|/(\epsilon^3(1-\gamma)^6)$. Moreover, V-MIN can afford larger values of ϵ and γ, compensating the errors with a slightly increased number of demonstrations.

Experimental Results

This section analyzes V-MIN performance:

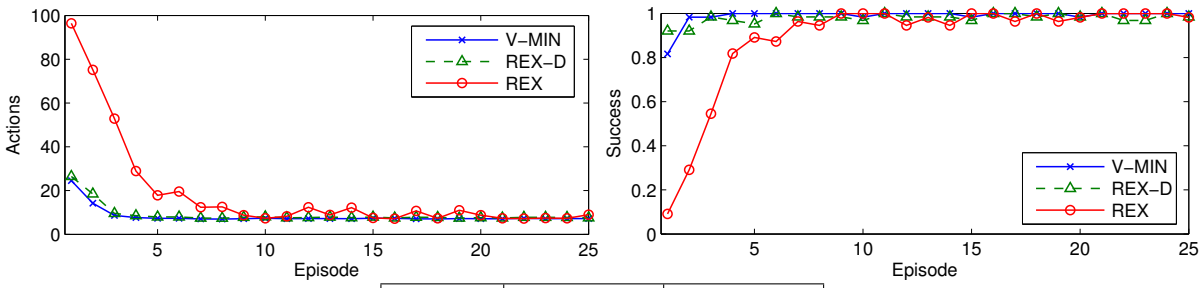
1. Comparing V-MIN with REX-D (Sec. 4.3) and REX (Lang et al., 2012).
2. Using different values of V_{min} and the increasing V_{min}.
3. Adaptation to changes in the tasks.

Episodes were limited to 100 actions before considering them as a failure. The exploration threshold was set to $\zeta = 3$, which yielded good results. Pasula et al. (2007)'s learner and Kolobov et al. (2012b)'s Gourmand planner were used. V-MIN and REX-D start with no previous knowledge, while REX has the list of actions available as it cannot learn them from demonstrations.

Experiment 1: Performance Comparison

In this experiment we compared V-MIN with REX-D and the R-MAX variant of REX in two problems of the International Probabilistic Planning Competition 2008. The V_{min} value was initialized with values slightly below the optimum.

Triangle Tireworld This is the *Triangle Tireworld* domain of the IPPC 2008 (Sec. 2.2.1), but with a 0.35 probability of getting a flat tire while it moves. Figure 4.12 shows that V-MIN always



	V-MIN	REX-D
Teacher	7.08 ± 2.65	5.67 ± 1.32

Figure 4.13: Experiment 1 (Exploding Blocksworld, problem 5): Means over 100 runs. Same layout as in Fig. 4.12.

learns the safe policy, as it is the only one with $V(s) > V_{min}$. However, the other approaches fail to learn the safe path when no flat tires are obtained during the first episodes: once they consider the short path as known, they never try to find better paths. In contrast V-MIN looks for alternatives with higher values after discovering the dead-ends in the short path. Therefore, V-MIN requests a few more demonstrations but finds the best path in terms of safety. Note that the large number of action executed by REX is due to its inability to detect dead-ends, so it keeps trying up to the 100 action limit.

This problem has 3 actions and 5 predicates with 43 and 50 respective groundings. Based on the performance analysis in Sec. 4.5.1, using a discount factor $\gamma = 0.9$, an accuracy parameter $\epsilon = 0.1$ and an exploration threshold $\zeta = |S|/(\epsilon^2(1-\gamma)^4) = 3$, the upper bound for the R-MAX sample complexity is proportional to $|S||A|\zeta/(\epsilon(1-\gamma)^2) = 50 \cdot 43 \cdot 3/(0.1(1-0.9)^2) = 2.2 \cdot 10^6$. Meanwhile, the estimated V-MIN sample complexity is $K_{\mathit{eff}} \log_2(|S|)|R|\zeta/(|S| \cdot \epsilon(1-\gamma)^2) = 2\log_2(50) \cdot 6 \cdot 3/(50 \cdot 0.1(1-0.9)^2) = 4 \cdot 10^3$, which is much lower. Note that V-MIN can use such a small ζ as it compensates the lack of exploration with a few extra demonstrations. Experimental results were better than this worst case, requiring just 14 exploratory actions and 3.87 demonstrations.

Exploding Blocksworld The results obtained in the *Exploding Blocksworld* domain (Sec. 2.2.1) are displayed in Fig. 4.13. This is a good example of a domain with a simple goal. Once the needed actions are demonstrated, exploration is enough to learn policies that yield the highest value. As REX-D already performs well, V-MIN obtains similar results. V-MIN requests a slightly larger number of demonstrations, and in exchange has a slightly higher success ratio and explores less. REX also converges to the optimal policy, but requires an average of 250 extra exploratory actions compared to the few demonstrations requested by the other two algorithms.

Experiment 2: V-MIN Adaptation Capabilities

This experiment was performed in a simulated version of the *Table Clearing* task (Sec. 2.2.2). The reward is based on the time spent, but there is also a high penalty for each broken object. Each action reduces the reward by 0.05, while the *moveStacks* action reduces it by 1 for every stack, by 5 for every broken object, and increases it by 5 when it finishes clearing the table. V-MIN requires the reward function to be known, and therefore the reward yielded by *moveStacks* is known since the beginning, while all other actions are just demonstrated.

Increasing V-MIN In this experiment the table has 3 plates and a cup on it, and there is one fork placed on one of the plates. The performance of V-MIN is evaluated for different values of V_{min}, and the increasing V_{min}, that starts at $V_{min} = 1.2$ and increases by one at the episodes 21 and 41.

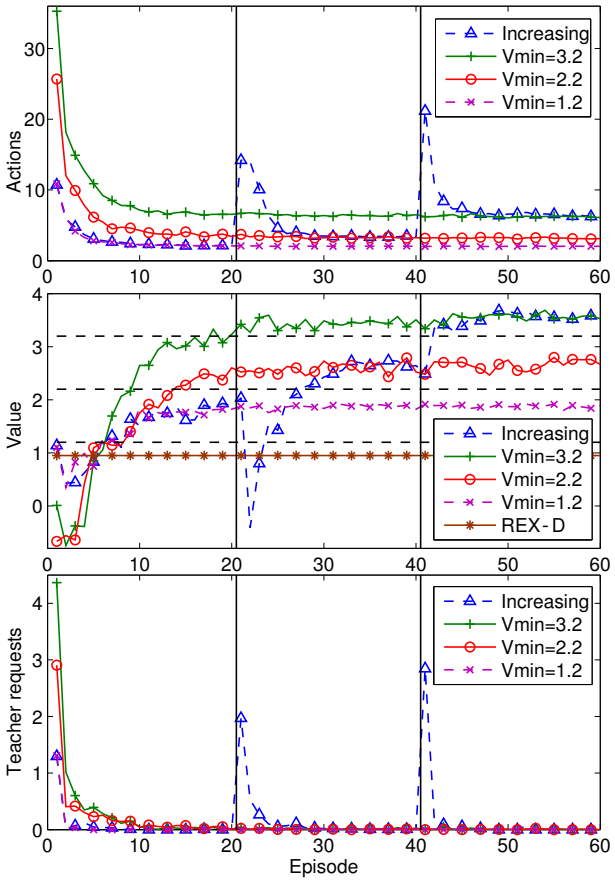
Figure 4.14 shows the results. REX-D is goal-driven and thus it just takes the stacks to the kitchen without trying to improve its policy. In contrast, V-MIN learns until it surpasses the V_{min} threshold. Higher V_{min} values increase the problem complexity, requiring more exploration and demonstrations, as extra actions are needed to get such values. Note that the learning is concentrated during the first episodes.

Finally, the increasing V_{min} learns incrementally at the pace requested by the teacher. V-MIN maintains its previous knowledge, and thus the exploration and demonstrations required to incrementally get to a certain V_{min} is similar to learning that V_{min} directly. When no new dead-ends have to be learned, as happens when changing from $V_{min} = 2.2$ to 3.2, the value also smoothly increases without significant degradations due to exploration.

Task Changes This experiment shows the V-MIN capabilities to adapt to changes in the scenario. Knowledge is maintained in all configurations but the "$V_{MIN} = 3.2$ reset" one, which starts each problem with no previous knowledge. Three different problems are solved:

1. The table has 3 plates and a cup on it.
2. Same as 1, but the cup is placed on a plate instead.
3. Same as 1, but there is a new fork on one of the plates.

Figure 4.15 shows the results. For the first two problems, the set of actions needed to get a value of $V^{\pi}(s) \geq 1.2$ is already enough to find policies that get $V^{\pi}(s) \geq 2.2$, and thus both yield the same results. The adaptation to problem 2 is very smooth, as the only change is that a cup starts placed on a plate, and just in the case of $V_{min} = 3.2$ a new action "removeCup" is required if it was not already learned. Note that "$V_{MIN} = 3.2$ reset" has to learn everything



$V_{MIN} = 1.2$	$V_{MIN} = 2.2$	$V_{MIN} = 3.2$	Increasing
1.39 ± 0.62	5.34 ± 2.21	8.09 ± 2.50	8.18 ± 2.01

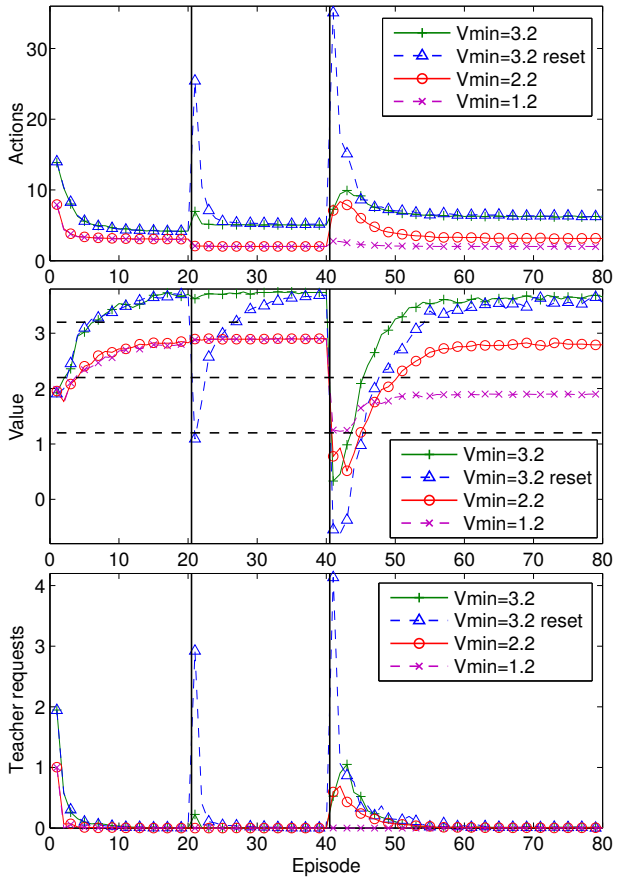
Figure 4.14: Experiment 2 (Table Clearing, standard): Means over 100 runs. The vertical black lines indicate that V_{min} was increased by 1. **Top:** actions executed per episode. **Upper middle:** value per episode. **Lower middle:** teacher demonstrations per episode. **Bottom:** total number of demonstrations requested.

again with a slightly more complicated initial state, resulting in a larger number of actions and demonstrations.

In contrast, problem 3 adds a new type of object, requiring to learn new actions and refine previous rules. V-MIN adapts rather well in just a few episodes, meanwhile "$V_{MIN} = 3.2$ reset" needs a much larger number of actions.

Conclusions

Learning with real robots is very time consuming if actions take several seconds to complete. Therefore, learning algorithms should require as few executions as possible. We addressed this



$V_{MIN} = 1.2$	$V_{MIN} = 2.2$	$V_{MIN} = 3.2$	Reset
1.13 ± 0.39	4.46 ± 1.34	8.68 ± 2.19	16.6 ± 5.73

Figure 4.15: Experiment 2 (Table Clearing, task changes): Means over 500 runs. The vertical black lines indicate the problem changes. Same layout as in Fig. 4.14.

problem using relational RL with demonstrations. We combined the generalization capabilities of learning in relational domains, where objects of the same type have to be learned only once, with teacher demonstrations, thereby significantly reducing the number of exploratory actions required, which is the longest part of the learning process.

We have proposed the REX-D algorithm that adds demonstration requests to the relational RL algorithm REX. In addition to faster learning, demonstrations allow the algorithm to learn new actions and adapt easily to different tasks. Moreover, REX-D can be integrated with any model-learner and automated planner, and the expressibility of the model learned will depend on the learner and planner being used. We proved experimentally that REX-D could learn domains with uncertain actions requiring much fewer exploration than previous approaches, and it was also able to adapt to similar problems very quickly.

Another aspect of REX-D is the use of action rule analysis to help the teacher, which minimizes the number of demonstrations required in a task. The improvements include providing guidance to the teacher so only the unknown actions need to be demonstrated, checking for valid action rule alternatives before requesting new demonstrations, and using subgoals in tasks where demonstrating new actions may require the execution of previously known actions.

An approach to avoid dead-ends has also been presented. It extends the REX-D algorithm to reason about dead-ends, analyze the causes, avoid them in the easier cases, and interact with a teacher to solve more complex ones. Experimental results showed that success ratios could be greatly improved with just a few extra teacher interactions.

Finally, we have proposed the novel V-MIN algorithm that learns policies satisfying the teacher quality requirements. The V_{min} parameter gives the option of either learning good policies at the cost of a large number of demonstrations and exploratory actions, or being more permissive with worse policies that are easier to learn. Even if the right V_{min} is not selected in the beginning, it can be tuned online until the teacher expectations are satisfied. As a result, V-MIN provides a generalization of REX-D that is not limited to goal-driven domains and allows the user to tune the amount of demonstrations being requested and the quality of the learned model.

5

Learning Models for Planning

Traditionally, applications have mostly used deterministic planners (Hoffmann and Nebel, 2001; Helmert, 2006) as they are easier to set-up and can solve complex problems with many variables. Recently the planning community has improved successfully the existing planners to solve more expressive and complex tasks. In the latest International Probabilistic Planning Competition (IPPC 2014) (Vallati et al., 2015), the best planners were able to solve probabilistically interesting tasks (Little and Thiebaux, 2007b) that included both action effects and exogenous effects (Kolobov et al., 2012a; Keller and Eyerich, 2012) and many relational variables. We consider that action effects are those that occur when the agent executes an action (e.g. a robot moves to a new position, the robot grasps an object), while exogenous effects are those that do not depend on an action (e.g. people moving in the street, a traffic light turns red, a plant grows).

However, such tasks with exogenous effects could not be learned with the approaches presented in the previous chapter, as they used Pasula et al. (2007)'s learner to obtain the model. To tackle more expressive models, we created a new learner that could obtain relational probabilistic models with both action effects and uncertain effects from a log of completely observable state-action-state experiences. This chapter presents this model learner, describes how it improves over previous state-of-the-art learners, and shows that it can be integrated in a RL approach to solve an unknown task with exogenous effects autonomously.

Finally, we will present how to use the proposed learner with a robot to improve its performance in tasks where exogenous effects are important. Specifically, we show a robot that has to help clearing the tableware on a table. Exogenous effects allow us to model the frequency at which different types of tableware arrive, and to coordinate with the waiter robots that take piles of tableware back to the kitchen. When a model with such exogenous effects can be learned, the performance of the task is increased significantly.

To summarize, we propose a novel method that takes as input a set of state-action-state

experiences, and learns a relational model with probabilistic and exogenous effects to be used for planning. We also show that the learner can be integrated in a RL framework to learn new tasks autonomously. Apart from simulated tasks, the approach will be used by a robot decision-maker to improve the performance in the task of clearing the tableware of a table.

This work appears in the following publications: (Martínez et al., 2016b, 2015d, 2017).

Previous Work

There exist previous works that tackle the problem of learning models. Some of them estimate the parameters of a model (Moldovan et al., 2012; Thon et al., 2011), but the model has to be provided and only its parameters are learned. The work by Jiménez et al. (2008) can capture the preconditions and uncertainty in the models given that the possible effects are known in advance. A more complete approach by Sykes et al. (2013) learns probabilistic logic programs, but the restrictions for the initial set of candidate rules need to be manually coded. In contrast, we learn the complete model and no restrictions are required.

Most approaches that learn complete models handle deterministic tasks, and although they can tackle partial observability (Mourão et al., 2012; Zhuo and Kambhampati, 2013) or apply transfer learning (Zhuo and Yang, 2014), they do not consider uncertain effects. In this work we focus on models with uncertain effects.The most similar approaches to ours are those that learn relational action models with uncertain effects (Pasula et al., 2007; Deshpande et al., 2007; Mourão, 2014). They learn the effects that each action may have for each set of preconditions, which are then represented with probabilistic STRIPS-like models. However, they do not learn exogenous effects that are not related to any action. These methods cannot be easily extended to learn exogenous effects. Pasula et al. (2007) and Deshpande et al. (2007)'s approach use a local search algorithm that works well when experiences are explained by one rule, but faces many local minima when tackling domains with exogenous effects, as two or more new rules have to be added to properly explain a experience. Mourão (2014)'s approach exhibits a similar problem since it learns one rule per experience. We take a different approach that learns models with uncertain effects, and also those with exogenous effects.

The problem of learning minimal effects from a log of input experiences is known to be NP-Hard (Walsh, 2010). The approaches shown before, as well as our approach, apply heuristics to find solutions with any number of input experiences. Optimal approaches that learn complete probabilistic models have also been proposed (Walsh et al., 2009), but they require too many input experiences.

Model Learner

In this section we show how to learn a relational probabilistic model with exogenous effects from a log of input state-action experiences. The learned model will consist of a set of planning operators (Sec. 2.1.1) that define both action effects and exogenous effects. The proposed method can be divided into two parts:

- *Candidate planning operator generation*. Candidates are generated with the LFIT (Learning From Interpretation Transitions) framework (Inoue et al., 2014). LFIT induces a set of propositional rules that realize the given input experiences. Specifically, an algorithm that guarantees to learn the set of minimal rules is used (Ribeiro and Inoue, 2014).

- *Planning operator selection*. To select the best subset of candidates, we define a score function that is maximized by candidates that explain input experiences while being general enough. Based on this score function, a search optimization method guided by an heuristic function is proposed. Moreover, suboptimal solutions to make complex tasks tractable are provided.

Our approach combines (a) LFIT on the propositional level to ensure that candidates are minimal, (b) an optimization method that works on the relational level to apply relational generalizations when selecting subsets, and (c) grounded input data. Since, as mentioned, the approach requires three different types of data (grounded, relational and propositional), data transformation methods are needed.

Candidate Planning Operator Generation

The input of the method is a set E of training experiences which are triples $e = (s, a, s')$ where s' is a successor state of s when the action a was executed. The output is a set of planning operators $\mathcal{O}$ that define the model. Figure 5.1 shows the data transformations and processing done in algorithm, which are described below:

- Transform grounded experiences to relational (non-grounded) ones. The objective is to learn relational operators that can take objects as arguments to generalize. Note that grounded literals are different from propositional ones because the objects and the variables in their terms can be identified. The transformation to relational experiences requires to substitute objects with variables.

- Obtain candidate operators. LFIT is used to obtain all possible candidate propositional rules for a given set of experiences. The main advantage of using LFIT is that it obtains

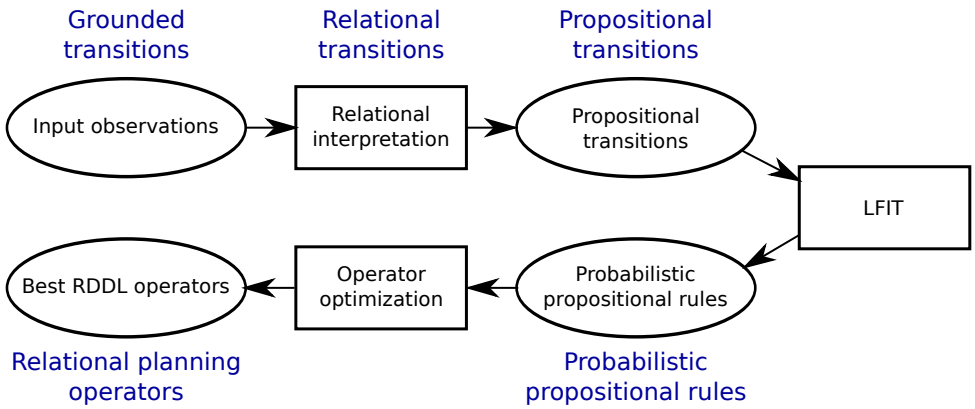


Figure 5.1: Data representation used for each module. The input and output data are shown in ellipses, and the processing modules are shown in boxes. The data representation used is indicated at each step.

the set of minimal rules that model both action effects and exogenous effects. To use LFIT, first the relational experiences have to be transformed to propositional ones, and later, the output propositional rules to planning operators.

- Select the subset of candidate planning operators that best models the training experiences. This is detailed in Section 5.2.2.

The aim of these transformations is to generalize better. The grounded input experiences could be straightforwardly transformed to propositional ones, but LFIT would learn rules that would model grounded data with no relational generalizations. When propositional experiences are obtained from relational ones, LFIT learns rules that model relational data. A relational representation is more compact and general as an infinite number of objects can be represented by each variable. The trade-off of learning relational generalizations is that the number of generated relational experiences is larger than the number of input grounded experiences, which increases the learning time.

Input Normalization

The first step is to normalize the input so that all state-action pairs (s, a) get the same number of experiences. We define that a experience $e = (s_e, a_e, s'_e)$ covers a pair (s, a) when $cov(e, s, a) = (s_e = s) \wedge (a_e = a)$ holds.

This step is specially important if the learner is integrated in a RL method, as state-action pairs with high expected values will be visited much often than those with low expected values. Without normalizing the input, the selection of the best operators (Section 5.2.2) would be biased.

The normalization is done by repeating experiences so that every (s, a) gets a similar number of covering experiences:

- *Input*: grounded experiences E.
- Get maximum coverage $n_{max} = \max_{(s,a)} |\{e \mid cov(e, s, a),\ e \in E\}|$.
- For each (s, a) covered by $E' \subset E$ (with $|E'| > 1$):
  - Number of times each experiences should be repeated: $n_{rep} = n_{max}/|E'|$.
  - Add n_{rep} times each $e' \in E'$ to E_{out}.
- *Output*: normalized experiences E_{out}.

Note that this input normalization does not have an impact on performance if implemented correctly. Every experience should have a counter representing the number of times it is repeated to avoid extra computation.

Grounded to Relational Experiences

The goal is to obtain a relational representation that can generalize to different objects. If the dynamics of an object are learned, the same dynamics can be applied to other objects, not requiring examples of every possible grounding.

Since generating all possible relational combinations of every experience would be highly inefficient, we limit the number of relational variables to a fixed number ω, which imposes a limit on the maximum number of variables that learned operators will have. Selecting the right value of ω is important. To learn effects that involve n objects, a value $\omega \geq n$ is required. However, the number of relational experiences scales exponentially with ω, thus a large value of ω is intractable.

This module generates all possible relational experiences with at most ω variables. For every experience $e \in E$, the following method is applied:

- *Input*: grounded experience $e = (s, a, s')$, max variables ω. We define the objects in s as $b_{s,i}$ and the objects in a as $b_{a,i}$. The action a has m objects.
- Obtain combinations of ω objects. For each combination of $\omega - m$ objects $(b_{s,1}, \dots b_{s,\omega-m})$ that are not in the action $(b_{s,i} \neq b_{a,j}, \forall\, i, j)$ do:
  - Create $v_{obj} = (b_{a,1}, \dots b_{a,m}, b_{s,1}, \dots b_{s,\omega-m})$ where $(b_{a,1}, \dots b_{a,m})$ are the objects in the action a.
  - Add v_{obj} to V_{obj}.

- For each $v_{obj} \in V_{obj}$:
  - A new experience $e' = e$ is created.
  - Replace in e' all objects in v_{obj} for variables.
  - Remove from e' any remaining literal with objects.
  - Add the new experience e' to E'.
- *Output*: a set of relational experiences E'.

Example 5.1. Given a grounded experience $(s, a \rightarrow s')$:

$$at(r1) \wedge road(r2, r3) \wedge road(r1, r3), move(r3) \rightarrow$$
$$road(r1, r3) \wedge road(r2, r3) \wedge at(r3),$$

the following relational experiences are generated (ω=2):

$$\begin{aligned} v_{obj} = (r3, r1) :\ & at(?Y) \wedge road(?Y, ?X), move(?X) \rightarrow \\ & road(?Y, ?X) \wedge at(?X); \\ v_{obj} = (r3, r2) :\ & road(?Y, ?X), move(?X) \rightarrow \\ & road(?Y, ?X) \wedge at(?X). \end{aligned}$$

Relational to Propositional Experiences

To create the input that LFIT requires, which are pairs (s, s') of propositional states, a library L_{conv} that converts between relational and propositional atoms is created. For each relational atom d_r, a new propositional atom d_p is created and the pair (d_r, d_p) is added to L_{conv}. Using L_{conv}, everything is substituted by its propositional counterpart:

- Relational literals are represented with propositional atoms that take the values 1 (true) or 0 (false).
- Relational experiences are triples (s, a, s'), while propositional experiences are pairs (s, s'). Therefore, an additional multi-valued atom is added to propositional experiences to represent the action. This atom takes as value the corresponding action name in L_{conv}, or "*noaction*" if there is no action. Note that the multi-valued representation is currently only used to model this action atom: other literals are binary, but there may be several different actions (and only one action per experience).

Example 5.2. Using the relational experiences in example 5.1, the following library is created $L_{conv} = \{(at(?Y), b1),\ (at(?X), b2),\ (road(?Y, ?X), b3),\ (move(?X), b4)\}$. The propositional experiences obtained by using L_{conv} are:

$$(b1{=}1) \wedge (b3{=}1) \wedge (action{=}b4) \rightarrow (b3{=}1) \wedge (b2{=}1);$$
$$(b3{=}1) \wedge (action{=}b4) \rightarrow (b3{=}1) \wedge (b2{=}1).$$

LFIT

The LFIT framework (Inoue et al., 2014) is used to obtain the set of probabilistic candidate rules that model the dynamics. Given a batch of propositional experiences (s, s'), LFIT induces a *normal logic program* that realizes the given experiences. This program consists of propositional rules that take the form:

$$r = m_h : p(r) \ \leftarrow \ m_1 \wedge \cdots \wedge m_n \tag{5.1}$$

where m_h is the head of the rule, $p(r)$ the probability, and $m_1 \wedge \cdots \wedge m_n$ is the body of the rule.

This framework has been extended (Ribeiro and Inoue, 2014) with a new algorithm that guarantees that the learned rules are minimal: the body of each rule constitutes a prime implicant to infer the head. It is based on a top-down method that generates hypotheses by *specialization* from the most general rules. Moreover, the framework has been adapted to capture also non-deterministic dynamics (Martínez et al., 2015d) . Our approach uses the specialization and non-deterministic algorithm of LFIT, so it learns the set of minimal probabilistic rules that models all effects appearing in the input experiences. It learns both action effects and exogenous effects because the action is just another atom that may or may not appear in the body of a rule.

Propositional Rules to Planning Operators

Planning operators (Eq. 2.2) can be reconstructed from probabilistic rules (Eq. 5.1) by using the library L_{conv} created before. For each propositional rule:

- The atoms in the body and head of the rule are transformed to relational ones (using L_{conv}), and added to the body and head of a planning operator.

- The action is extracted from the multi-valued action atom in the rule body. If the atom value is not "*noaction*", the corresponding action in L_{conv} is added to the operator.

Example 5.3. Given that LFIT had learned the following rule

$$(b2{=}1) : 0.8 \leftarrow (b1{=}1) \wedge (b3{=}1) \wedge (action{=}b4),$$

using the library L_{conv} generated in example 5.2, the resulting operator $o(?X, ?Y)$ is

$$at(?X) : 0.8 \leftarrow at(?Y) \wedge road(?Y, ?X), move(?X).$$

Traditionally, PPDDL (Younes and Littman, 2004) has been the standard language to model probabilistic domains, but it is difficult to model exogenous effects with it. Therefore, our approach uses the RDDL language (Sanner, 2010), which has been the standard for the latest probabilistic planning competitions (IPPC 2011 and 2014). Writing our planning operators with RDDL is straightforward, and this language can be used directly by state-of-the art planners. RDDL objects and variables have types, and a variable can only be substituted by an object of the same type. However, for clarity and simplicity, we assume through the chapter that there are no types, as adding them is trivial.

Planning Operator Selection

LFIT provides the set of minimal rules (that have been transformed to planning operators) that describe all the experiences. Note that LFIT learns the set of minimal rules, and not the minimal set of rules, so many operators may model the same changes and underfit or overfit. The subset of planning operators to model the experience dynamics is selected as follows:

- A score function is defined to evaluate the operators.
- A heuristic search algorithm selects the set of operators that maximizes the score. Note that this set may differ from the actual model, as it depends on the coverage of the input experiences and the quality of the score function.
- The subsumption tree is used to improve efficiency by partitioning the set of candidates into smaller subsets.

Score Function

The score function values the quality of a set of operators. The following functions are used by the score function:

- The likelihood is calculated using Eq. 2.5.
- The penalty term $Pen(\mathcal{O}) = \sum_{o \in \mathcal{O}} |body(o)|$ is the number of atoms in the operator bodies.
- The confidence $Conf(E, \epsilon)$ is obtained from Hoeffding's inequality. The probability that an estimate $\widehat{o_{prob}}$ is accurate enough $|\widehat{o_{prob}} - o_{prob}| \leq \epsilon$ with a number of samples $|E|$ is bounded by $Conf(E, \epsilon) \leq 1 - e^{-2\epsilon^2|E|}$.

Finally, the proposed score function is defined as

$$s(\mathcal{O}, E) = \underset{e \in E}{E}[\log(P(e|\mathcal{O}))] - \alpha \frac{Pen(\mathcal{O})}{Conf(E, \epsilon)}, \tag{5.2}$$

where $\alpha > 0$ is a scaling parameter for the penalty term. This score function is based on Pasula et al. (2007)'s one, where the likelihood is maximized to obtain operators that explain the experiences well, and the penalty term is minimized to prefer general operators when specific ones have very limited contributions. In contrast to Pasula et al.'s approach, the confidence term is added so that the penalty is increased when few experiences are available, as the estimates are less reliable.

Heuristic Search

Given a set of operators $\mathcal{O}$ with the same head, a heuristic search method is used to find the best subset of operators that maximizes the score function. To that end, we define the heuristic version of the change likelihood (Eq. 2.4) as:

$$P_h(c|\mathcal{O}) = \begin{cases} P(c|o_g), & \exists! o_g \in Gr(\mathcal{O}) \,|P(c|o_g) > 0 \\ 1 - \delta, & \nexists\, o_g \in Gr(\mathcal{O}) \,|cov(o_g, s, a) \\ 0, & \text{otherwise } (|Gr(\mathcal{O})| > 1), \end{cases} \tag{5.3}$$

where δ is a parameter that can trade quality for efficiency. This heuristic modifies the change likelihood (Eq. 2.4) when no operator covers the change, giving a likelihood of $1 - \delta$ instead of 0. The heuristic score function $s_h(\mathcal{O}, E)$ is defined as the score function (Eq. 5.2) but replacing the standard change likelihood (Eq. 2.4) with this heuristic likelihood.

This heuristic gets the expected likelihood that can be obtained by adding new operators to $\mathcal{O}$. When $\delta = 0$, it works as an admissible heuristic (prop. 5.1) as it gives the maximum likelihood $= 1$ to uncovered changes. When $\delta > 0$ but close to 0, then the heuristic penalizes very specific operators when more general operators with a high likelihood are also available. The practical result is that the algorithm usually runs faster, but the heuristic is not admissible anymore.

Algorithm 8 selects the best subset of operators to explain the input experiences. In line 1, the candidate list Γ is initialized by creating one separate subset for each operator in the input set of candidates $\mathcal{O}_{input}$. Note that Γ is a set of sets of planning operators, which is initialized to $\Gamma = \{\{o_1\}, \ldots, \{o_n\}\}$ assuming that $\mathcal{O}_{input} = \{o_1, \ldots, o_n\}$. Afterwards, lines 2-3 find the best subset in Γ (which is the best set with only one operator). From that point, the candidate sets in Γ are iteratively joined together to find the best set with any number of operators, until none has

Algorithm 8 OperatorSelection($\mathcal{O}_{input}$, E)

1: Current candidates $\Gamma \leftarrow \{o\}, \forall\, o \in \mathcal{O}_{input}$
2: $max_{score} = \max_{\mathcal{O} \in \Gamma} s(\mathcal{O}, E)$
3: $\mathcal{O}_{best} = \operatorname{argmax}_{\mathcal{O} \in \Gamma} s(\mathcal{O}, E)$
4: **while** $\max_{\mathcal{O} \in \Gamma} s_h(\mathcal{O}, E) > max_{score}$ **do**
5: $\quad \mathcal{O} = \operatorname{argmax}_{\mathcal{O} \in \Gamma} s_h(\mathcal{O}, E)$
6: $\quad$ Remove $\mathcal{O}$ from Γ
7: $\quad$ **for** $\mathcal{O}' \in \Gamma \mid \texttt{IsNew}(\mathcal{O} \cup \mathcal{O}')$ **do**
8: $\quad\quad \mathcal{O}_{new} = \mathcal{O} \cup \mathcal{O}'$
9: $\quad\quad$ **if** $s(\mathcal{O}_{new}, E) > max_{score}$ **then**
10: $\quad\quad\quad max_{score} = s(\mathcal{O}_{new}, E)$
11: $\quad\quad\quad \mathcal{O}_{best} = \mathcal{O}_{new}$
12: $\quad\quad$ **end if**
13: $\quad\quad$ Add $\mathcal{O}_{new}$ to Γ
14: $\quad$ **end for**
15: **end while**
16: **Output** $\mathcal{O}_{best}$

$s_h(\mathcal{O}, E) > max_{score}$ (lines 4-15). In lines 5-6, the candidate $\mathcal{O}$ with the largest heuristic score is selected and removed from Γ. Then, in lines 7-8, new candidates are generated by combining the selected subset $\mathcal{O}$ with every subset in Γ. The *IsNew* method checks that the new candidate has not been already analyzed. If any of the new candidates has a new best score, it is saved as the best candidate (lines 9-11). Finally, the new candidates are added to Γ.

This method works as a search algorithm guided by an heuristic. The nodes to be analyzed are the subsets of operators stored in Γ, where they are ordered by the heuristic score value. The search tree is expanded by joining one subset with every other subset. Finally, the algorithm continues until no subset has a heuristic score larger than the best score so that the solution has been found.

The search can be used as an *anytime* algorithm, it can be stopped at any point to get the best solution found so far. Moreover, there are two options to limit in advance the processing time of the algorithm in complex problems: set a time limit, or set a limit in the size of Γ (only maintain the κ sets with the highest score in Γ). Experimental tests showed that in some domains the heuristic leads quickly to the best set, and subsequent processing is done only to confirm that no other set is better.

Property 5.1. If $\delta = 0$, then the heuristic s_h is admissible: $\forall \mathcal{O}, s_h(\mathcal{O}) \geq s(\mathcal{O}^*) \mid \mathcal{O}^* = \operatorname{argmax}_{\mathcal{O}' \supset \mathcal{O}} s(\mathcal{O}')$. Therefore the optimal set will be found.

The two parts of the score function in Eq. 5.2 (likelihood and regularization) can be analyzed

separately. Note that subsets of operators may only increase in size, as they start with one operator and can only be combined with other subsets.

- When adding operators to a set, the likelihood only increases when experience changes that were not covered before are covered by the added operators. In the heuristic score (Eq. 5.3) all non-covered experience changes are already set to the maximum value of 1, so adding new operators cannot improve the result over the heuristic.
- The regularization part of the score function (which is $Reg(\mathcal{O}) = -\alpha\frac{Pen(\mathcal{O})}{Conf(E,\epsilon)}$) is monotonically decreasing as $Pen(\mathcal{O})$ can only increase when adding more operators, and $Conf \in (0, 1]$, $\alpha > 0$, $Pen \geq 0$.

Property 5.2. When relaxing the admissibility criterion with $\delta > 0$, the solution found by Algorithm 8 is bounded to be no worse than $\mathcal{C} \cdot \log(1 - \delta)$ plus the optimal score, where $\mathcal{C}$ is the average number of literals with the same predicate that change in a experience.

Operators with different head predicates are analyzed separately as their effects are independent, so $\mathcal{C}$ only depends on the average changes to one predicate literals. Also note that $\delta \in [0, 1)$, and thus, $\log(1 - \delta) \leq 0$. Let $\mathcal{O}_n$ be the optimal solution with n operators and $s(\mathcal{O}_n) = opt$, then $\forall i < n$, at least one predecessor of i operators $\mathcal{O}_i$ and $s_h(\mathcal{O}_i) \geq \mathcal{C} \cdot opt + \log(1 - \delta)$ will exist.

- The regularization term is monotonically decreasing (see explanation of property 5.1), so $Reg(\mathcal{O}_n) \leq Reg(\mathcal{O}_i)$.
- The maximum difference between $P(\mathcal{O}_n)$ and $P_h(\mathcal{O}_i)$ is $P(e|\mathcal{O}_n) = 1$ and $P_h(e|\mathcal{O}_i) = (1 - \delta)^x$, which is the case where $\mathcal{O}_n$ has perfect coverage, $\mathcal{O}_i$ has no coverage (all the coverage is obtained from $\mathcal{O}_n \setminus \mathcal{O}_i$) and the experience has x changes. Then, if we take the worst case for all experiences, and an average of $\mathcal{C}$ changes per experience, $P(\mathcal{O}_n) - P_h(\mathcal{O}_i) = E[\log(P(E|\mathcal{O}_n))] - E[\log(P_h(E|\mathcal{O}_i))] = \log(1) - \log(1 - \delta)^{\mathcal{C}} = \mathcal{C} \cdot \log(1 - \delta)$.

Therefore the predecessors of the optimal solution will be checked (and thus the optimal solution found) unless a solution with $s(\mathcal{O}) \geq opt + C \cdot log(1 - \delta)$ is found before.

A value of $\delta > 0$ can speed up the algorithm considerably at the expense of optimality. When two operators have similar likelihoods, $\delta > 0$ penalizes the most specific one. This results in general operator sets being analyzed first, and thus models with better likelihoods are obtained earlier.

Subsumption Tree

In this section we present a method to speed up the approach by partitioning the set of candidates into smaller groups. The idea of the subsumption tree is to start with the best subset of specialized operators, and iteratively check if more general operators yield better scores.

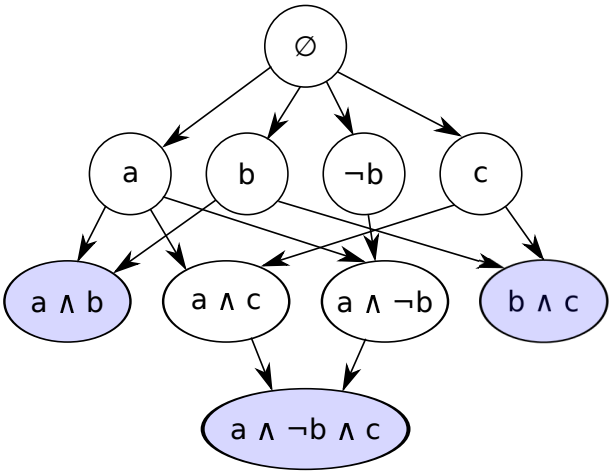


Figure 5.2: Example of a subsumption tree. Each letter (a, b, c) represents a literal in the extended body of a planning operator. The leaves are the nodes painted in blue.

Definition 5.1 (Subsumption relation)**.** First, we define the extended body of an operator as $body_{ext}(o) = body(o)(\wedge a)$ where $(\wedge a)$ only appears if o has an action. Let o_1 and o_2 be two planning operators with $head(o_1) = head(o_2)$, o_1 is subsumed by o_2 if $(body_{ext}(o_1) \supseteq body_{ext}(o_2))$.

Definition 5.2 (Subsumption tree)**.** The subsumption tree $Tree_{\mathcal{O}}$ of a set of planning operators $\mathcal{O} = \{o_1, ..., o_n\}$ is a directed graph with arcs (o_i, o_j) when o_i subsumes o_j and $|body_{ext}(o_j)| - |body_{ext}(o_i)| = 1$. We call the set of leaves $L(Tree_{\mathcal{O}})$. Figure 5.2 shows an example of a subsumption tree.

The subsumption tree orders the operators in levels that represent the generality of the operators: the less literals the more general the operator. Based on this tree, Algorithm 9 selects the operators. The idea is to start by identifying the best specific operators, and then check if their generalizations improve the results. In lines 2-4, the best subset of leaves $\mathcal{O}_{L,best}$ is identified, and all leaves not in $\mathcal{O}_{L,best}$ are removed from the tree. Then, in lines 5-7, a new set of operators $\mathcal{O}_P$ is created that includes $\mathcal{O}_{L,best}$ and the operators that subsume them (their parents in the tree). The best subset $\mathcal{O}_{P,best}$ in $\mathcal{O}_P$ is selected, and $\mathcal{O}_L \setminus \mathcal{O}_{P,best}$ are removed. This is repeated until nothing is changed in the tree.

The performance is improved by using the subsumption tree: it divides the candidates into subsets, and the planning operator selection is much faster with smaller sets of candidates. Although it sacrifices optimality, experimental tests showed that the results obtained were in many cases optimal or near optimal. This happens due to the fact that in most cases $P(\mathcal{O}) >> Reg(\mathcal{O})$. The operators in the leaves maximize $P(\mathcal{O})$ as they are more specialized, while the operators near the root maximize $Reg(\mathcal{O})$ as they are more general. Thus, the subset of leaves

Algorithm 9 OperatorSelectionSubsumption($Tree_{\mathcal{O}}, T$)

do
  $\mathcal{O}_L = leaves(Tree_{\mathcal{O}})$
  $\mathcal{O}_{L,best} = \texttt{OperatorSelection}(\mathcal{O}_L, T)$
  Remove $(\mathcal{O}_L \setminus \mathcal{O}_{L,best})$ from $Tree_{\mathcal{O}}$
  $\mathcal{O}_P = \mathcal{O}_{L,best} \cup parents(Tree_{\mathcal{O}}, \mathcal{O}_{L,best})$
  $\mathcal{O}_{P,best} = \texttt{OperatorSelection}(\mathcal{O}_P, T)$
  Remove $(\mathcal{O}_L \setminus \mathcal{O}_{P,best})$ from $Tree_{\mathcal{O}}$
while $Tree_{\mathcal{O}}$ changed
Output $= leaves(Tree_{\mathcal{O}})$

selected in the first iteration usually is near optimal, and afterwards the method only has to find the right level of generalization.

Note that if the subsumption tree is used, the best operators may be close to the root of the tree (and thus they will be analyzed at the end), so the learner shouldn't be used as an anytime algorithm. However, the processing time can still be bounded with satisfactory results by limiting the size of Γ to κ sets in Algorithm 8.

Experiments

This section describes the experimental evaluation of our approach. Three domains of the 2014 International Probabilistic Planning Competition (IPPC) (Vallati et al., 2015) were used in the experiments.

Domains

Three IPPC 2014 domains (Sec 2.2.1) were used in the experiments. Note that they were slightly modified to remove redundancy (e.g. a *north(?X,?Y)* literal is equivalent to *south(?Y,?X)*, so one can be replaced by the other).

- *Triangle Tireworld*. This domain is the easiest one, it has uncertain effects, but no exogenous effects. It is modeled with 5 different predicates, 3 actions, 7 operators, and operators require at most 2 terms ($\omega = 2$). This domain serves as a good baseline to compare with the state of the art as there are not exogenous effects.

- *Crossing Traffic*. This domain has an intermediate difficulty. It has uncertain effects and exogenous effects, which makes it more challenging, but the complexity of the model is still moderate: 8 predicates, 4 actions, 6 operators, and operators take at most 3 terms.

- *Elevators*. This is the most challenging domain. It has uncertain effects and exogenous effects. It is modeled with 10 predicates, 4 actions, 17 operators, and operators take at most 3 terms.

Evaluation of the Model Learner

To evaluate the learner the scheme used by Pasula et al. (2007) was followed. The learners had to obtain models from sets of input experiences $(s, a, s') \in E$ that were generated randomly. To create an experience, first the state s is constructed by randomly assigning a value (positive or negative) to every literal, but ensuring that the resulting state is valid (e.g. in the *Elevators* domain, an elevator cannot be in two different floors at the same time). Then, the action a arguments are picked randomly, and the state s' is obtained by applying all operators to (s, a). The distribution used to construct s is biased to guarantee that, in at least half of the examples, the operators that contain a have a chance of changing the state.

The evaluation of the learned models is carried out by calculating the *average variational distance* between the true model $\mathcal{O}$ and the estimate $\hat{\mathcal{O}}$. This evaluation uses a new set of similarly generated random experiences E':

$$D(\mathcal{O}, \hat{\mathcal{O}}) = \frac{1}{|E'|} \sum_{e \in E'} \left| P(e|\mathcal{O}) - P(e|\hat{\mathcal{O}}) \right|. \tag{5.4}$$

As the *average variational distance* may be difficult to interpret, here we give an intuition about the utility of the learned models. A planner can usually yield a plan when the *average variational distance* is below:

- 0.09 in the *Triangle Tireworld* domain.
- 0.15 in the *Crossing Traffic* domain.
- 0.1 in the *Elevators* domain.

As the *average variational distance* becomes lower, there is a higher probability that obtained plans are optimal.

We analyze experimentally the proposed algorithm, its parameters, and how it compares with the state of the art. The learner uses the following parameters: α, ϵ, and δ, which are the score function parameters; κ, which is the size limit of Γ (Algorithm 8); and "$tree$" to denote that the subsumption tree is being used.

The difficulty to learn a domain is mostly given by:

- The maximum number of terms ω that operators may have. The number of terms increases exponentially the number of relational experiences generated from the input grounded

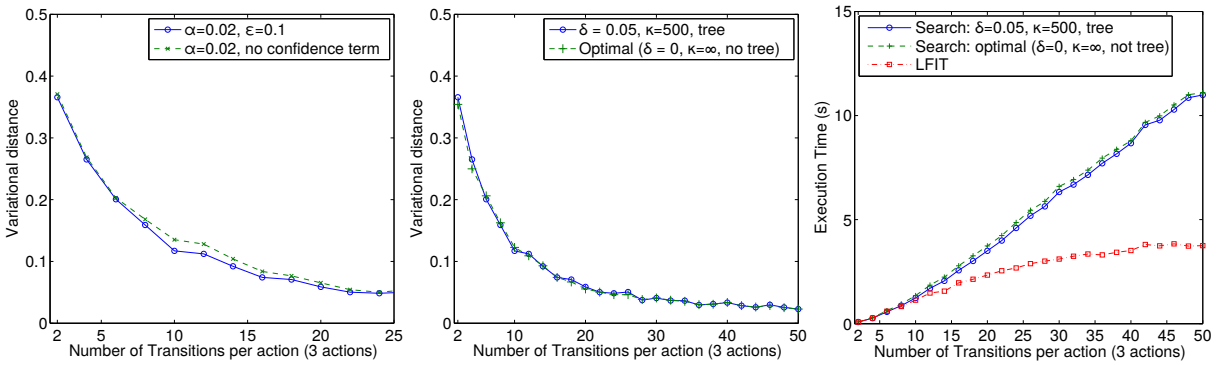


Figure 5.3: Evaluation of different configurations in the Triangle Tireworld domain, MDP-1. Unless stated otherwise, the following parameters were used: $(\alpha = 0.02,\ \epsilon = 0.1,\ \omega = 2,\ \delta = 0.05,\ \kappa = 500, \text{tree})$. The results shown are the means obtained from 100 runs. The evaluation was done with 3000 experiences. **Left:** Influence of the confidence term. **Center:** Comparing optimal and suboptimal search configurations. **Right:** Execution time of the search (optimization) and the LFIT modules. The total learning time would be the sum of LFIT plus one of the search configurations.

experiences (Section 5.2.1), and therefore the number of candidate operators. If the value of ω is larger than the number of terms that operators actually require, the learning time increases while the quality of the models remains the same.

- The number of literals, both constant and variable, used to represent the states. The candidates that LFIT generates consider all combinations of literals that are consistent with the experiences.

- The number of uncertain and exogenous effects. LFIT generates all candidates that may explain an effect, including operators that overfit and underfit, and all combinations of action effects and exogenous effects.

Configuration parameters

Here we discuss the impact of the different configuration parameters have on the quality of the models learned.

- As seen in figs. 5.3-left and 5.4-left, the confidence term in the score function improves the quality of the models learned when few input experiences are available. This confidence term penalizes very specialized operators in cases where there is a large uncertainty in the predictions. Once many experiences are available, the impact of this term disappears. Note that only probabilistic operators are improved as deterministic ones are completely specialized anyway. The impact of this term is relatively small in the experiments because both domains require only one probabilistic operator.

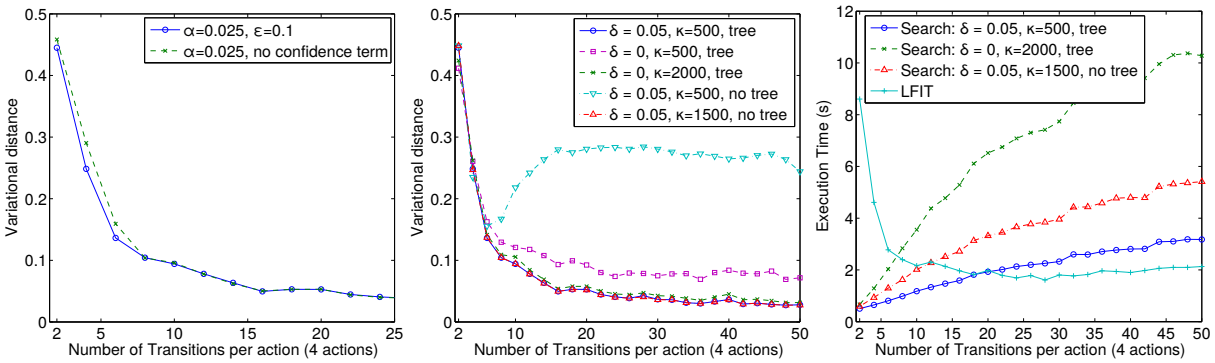


Figure 5.4: Evaluation of different configurations in the Crossing Traffic domain, MDP-1. Unless stated otherwise, the following parameters were used: $(\alpha = 0.025,\ \epsilon = 0.1,\ \omega = 3,\ \delta = 0.05,\ \kappa = 500, \text{tree})$. The results shown are the means obtained from 100 runs. The evaluation was done with 4000 experiences. **Left:** Influence of the confidence term. **Center:** Influence of δ and the subsumption tree. **Right:** Execution time of the search (optimization) and the LFIT modules. The total learning time would be the sum of LFIT plus one of the search configurations.

- Figure 5.3-middle shows how the optimal search configuration compares to one that uses the subsumption tree, a non-admissible heuristic ($\delta > 0$), and restricts the size of Γ to κ sets. Both yield almost the same results because the subsumption tree and the relaxed heuristic degrade the results only in very specific cases, and the value of κ is high enough to find the relevant operators.

  The *Triangle Tireworld* domain does not contain exogenous events, and thus, the selection of the best set of candidates is very simple and can be done equally fast in the optimal case (Fig. 5.3-right). Most of the execution time is spent in obtaining the likelihood of the operators as there are a lot of possible candidates in this domain.

- In contrast, when learning domains with exogenous events, the optimal search method was not used because the execution time was too large in all but the most simple problems. Similar results to the optimal method were obtained when the set of possible candidates Γ was restricted to a size κ that was large enough, and the execution time was greatly reduced.

  Figure 5.4-middle shows the advantages of the subsumption tree and a non-admissible heuristic ($\delta > 0$). The subsumption tree divides the search problem in smaller ones, and thus a lower κ is enough to yield good results. The same applies to the non-admissible heuristic since it prioritizes operators that explain many experiences with a high likelihood, and very specific operators that may not be interesting are given a smaller heuristic score. As a result, the configurations without the subsumption tree and $\delta = 0$ are slower because they analyze more operators to obtain the same results (Fig. 5.4-right).

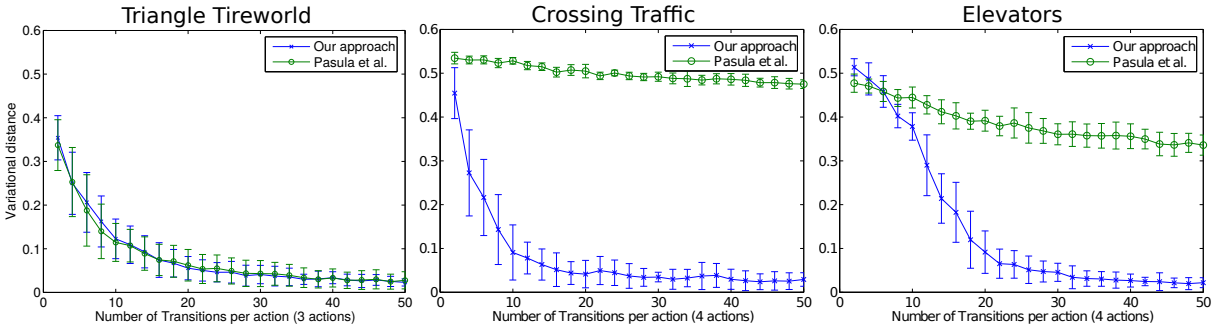


Figure 5.5: Comparison with Pasula et al. The results shown are the means and standard deviations obtained from 50 runs. The evaluation was done with 5000 random experiences. **Left:** Triangle Tireworld ($\alpha = 0.02$, $\epsilon = 0.1$, $\omega = 2$, $\delta = 0$, $\kappa = \infty$, no tree). **Center:** Crossing Traffic ($\alpha = 0.025$, $\epsilon = 0.1$, $\omega = 3$, $\delta = 0.05$, $\kappa = 500$, tree). **Right:** Elevators ($\alpha = 0.015$, $\epsilon = 0.1$, $\omega = 3$, $\delta = 0.05$, $\kappa = 500$, tree).

As an exceptional case, LFIT takes longer to execute in the *Crossing Traffic* domain with very few experiences as it finds more possible patterns. Its execution time increases again once many experiences have been added.

Comparison with Pasula et al.

Comparison is performed only with Pasula et al. (2007)'s learner, as Deshpande et al. (2007)'s learner is an extension to include transfer learning, and Mourão (2014)'s learner yields similar results to Pasula et al.'s approach in completely observable problems. The experiments were carried out with the implementation by Lang and Toussaint (2010).

Figure 5.5-left shows the results of learning the *Triangle Tireworld* domain. It can be easily learned by both, ours, and Pasula et al.'s approach, as there are no exogenous effects. With this experiment we want to show that our method can learn domains without exogenous effects as well as state-of-the-art learners.

Figures 5.5-center and 5.5-right refer to two domains with exogenous effects. As Pasula et al.'s learner cannot learn exogenous effects, it tries to build overcomplicated operators that explain both action effects and exogenous effects at the same time, and thus it is not able to yield a good general model. In contrast, our approach is able to distinguish action effects from exogenous effects once enough experiences are given as input.

Limitations of the Learner

The main limitation of the algorithm is the scalability. If the number of generated propositional atoms is large (either because the domain is represented with a large number of predicates, or because ω takes a high value), the problem may become intractable. The *Elevators* domain

has the highest complexity that our approach can learn within a reasonable time. When 25 experiences per action were provided, the planning operator selection took an average > 3 minutes.

Moreover, many IPPC 2014 domains cannot be directly learned. The following features would be needed to support all domains:

- The universal quantifier (*forall*) and the negative existential quantifier ($\sim$ *exists*) are not supported. When doing the translation from propositional rules to relational planning operators, new candidates could be generated that considered a positive atom in the preconditions as a (*forall*), or a negative atom as a ($\sim$ *exists*). It remains as future work to analyze the effectiveness of such candidates, and to check if the operator selection process would still be efficient with the addition of new candidates.

- Some IPPC domains have operators whose probabilities are encoded as a function. Unfortunately, our approach can only consider real numbers as probabilities.

Evaluation in RL

In this section we show the improvements of using the presented model learner in a RL algorithm instead of Pasula et al. (2007)'s learner. We chose V-MIN as the RL algorithm because it provided the best behavior (Sec. 4.5). Moreover, we decided to use the PROST planner (Keller and Eyerich, 2012) as it can obtain good results with probabilistic models containing exogenous effects.

Triangle Tireworld

It should be noted that in the IPPC 2014 representation of the *Triangle Tireworld* domain there is one exogenous effect: when the goal reward is received, the "goal-reward-received" literal becomes true. As this is modeled as an exogenous effect and Pasula et al.'s approach cannot learn it, that planning operator is given to it at the beginning. Our approach can learn this effect easily because it appears isolated after the goal is reached, so it doesn't have a significant impact on the performance.

Figure 5.6 shows the results of learning the *Triangle Tireworld* domain. Initially the robot chooses the shortest path and thus 50% of the times gets an irrecoverable flat tire. The learners have to learn that the robot cannot move with a flat tire, and that a spare tire is required to recover from a flat tire before the planner opts to go through the safe but longer path. Problem 2 is learned in less episodes because more actions are executed per episode. As expected, the performance of our learner is similar to Pasula et al.'s one. As both learners can obtain the

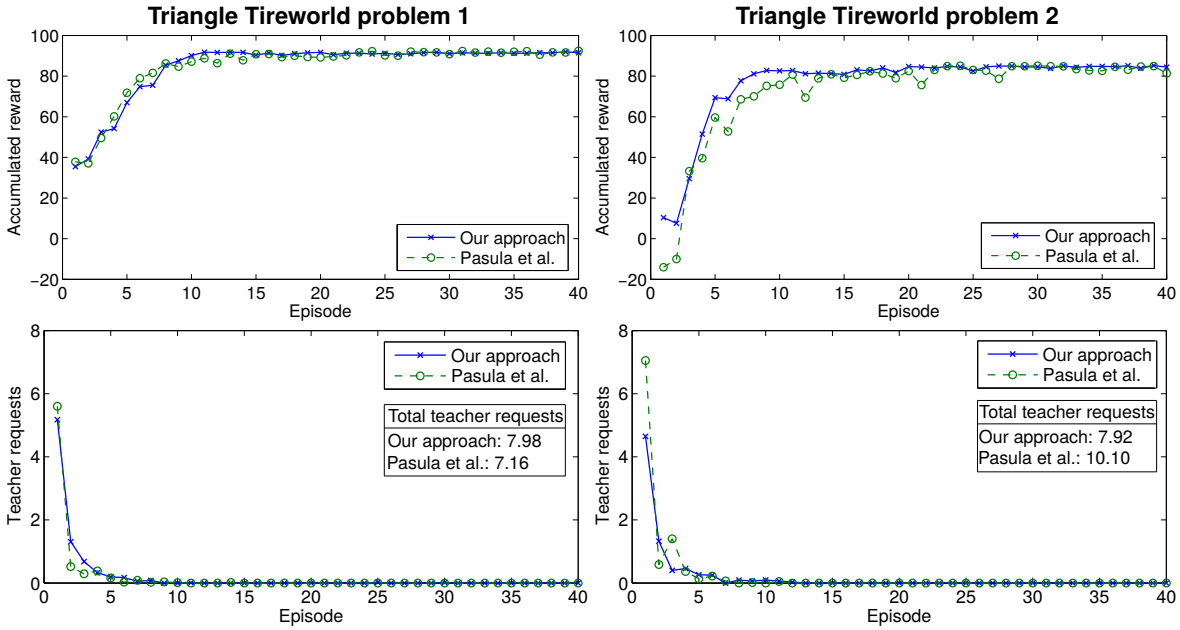


Figure 5.6: Learning the Triangle Tireworld domain with V-MIN. Comparison of using our approach and Pasula et al.'s one as the learner integrated in V-MIN. The difference between problem 1 and problem 2 is that in the latter the state space is larger, and more actions are required to reach the goal. The results shown are the means and standard deviations obtained from 100 runs. The exploration threshold is $\zeta = 3$ and V_{min} is 85 in problem 1, and 65 in problem 2.

model of the task with few experiences, they also work well when integrated in V-MIN. The only difference is that our approach learns more general models during the first episodes (because of the confidence term in the score function), and thus it works slightly better in problem 2, learning faster and requiring less teacher demonstrations. This is not an advantage in problem 1 because it is smaller and more specific operators also work well.

Crossing Traffic

The difficulty in this domain is learning that if the robot ends in the same position as a car, it will disappear independently of the action executed. Our model learner has to see one experience of the disappear effect for every action until it can learn the disappear effect completely. Figure 5.7 shows the results of the V-MIN learner. Pasula et al.'s learner was not compared here because, as mentioned previously, it cannot learn the exogenous effects in this task.

- Problems 1 and 2 are easier to learn, and they only require a few teacher demonstrations during the first episodes. In problem 1, the naive strategy of going directly to the goal has a high success ratio (70%), and thus it takes more episodes until the robot collides with

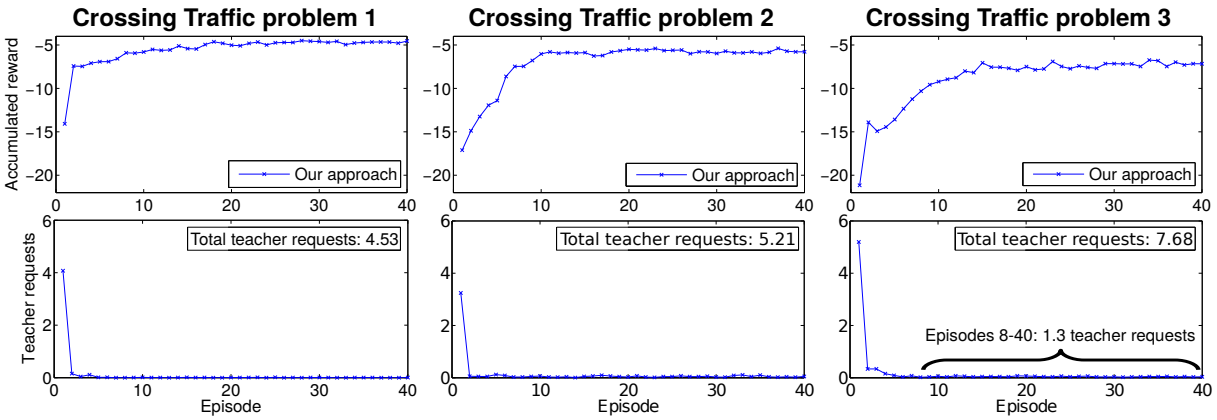


Figure 5.7: Learning the Crossing Traffic domain with V-MIN. The results shown are the means and standard deviations obtained from 300 runs. Problem 1 is the easiest set up: its state space is small and the probability that a car arrives to each middle lane is 0.3. Problem 2 has also a small state space but cars arrive with a probability of 0.6. Problem 3 has a larger state space and a 0.3 probability of car arrivals. The exploration threshold is $\zeta = 3$ and V_{min} is -8 in problem 1, -12 in problem 2, and -15 in problem 3.

a car and learns that it disappears. The robot will always take the shortest route until it learns that it disappears with every action.

- In problem 2, the success ratio of reaching the goal is only 40%, and thus it learns faster the disappear effect, and that it should avoid the cars.

- Problem 3 works similarly to problem 2 in the beginning as the probability of reaching the goal without dodging cars is around 50%. However, as the state space is larger there are special cases that only appear rarely. For example, the robot may start to go up, but if too many cars appear, and the robot cannot dodge them, it has to go back down by executing a "move-south" action. This action is not needed in problems 1 and 2, but it is needed in some cases in problem 3. After the 8th episode, most of the teacher demonstrations are requested to learn how to solve these unexpected problems that don't appear in the general case.

Robot Table Clearing

This section describes a task where a robot has to clear the tableware laying on a table. To that end, V-MIN with the learner presented previously was used as the decision-maker of the robot to solve the task.

This task represents a robotized restaurant. The manipulator robot that we control has to clear the tableware on a table. It has to cooperate with mobile robots that can take a pile of

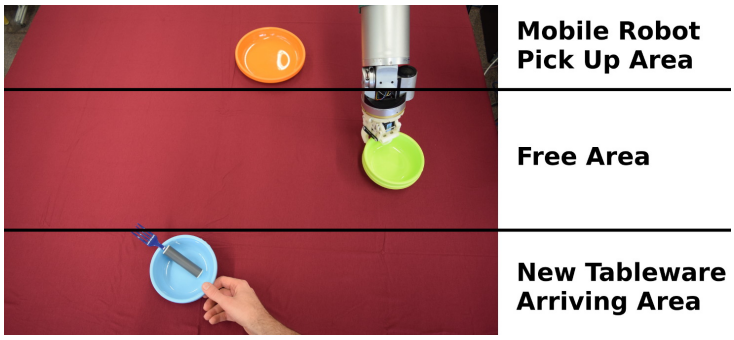


Figure 5.8: Table clearing task. On the bottom area, people leave the used tableware. On the top area, the robot has to prepare piles for the mobile robots that take tableware to the kitchen. The middle space is free for the robot to store objects while piling them up and waiting for mobile robots.

tableware from the table to the kitchen to be cleaned. The number of mobile robots is limited, so our robot should pile tableware together to minimize the number of piles to be taken. The difficulty of the task comes from the fact that people will continuously bring new used tableware to the table. The robot has to organize continuously the tableware on the table so that there is always enough space for people to leave new tableware, and also to ensure that the mobile robots always have a prepared pile of tableware when they come to the table.

This task is illustrated in Fig. 5.8. People leave used tableware on the bottom area, the robot has the central area to organize tableware and create piles, and the top area is where the mobile robots pick a pile up when they come to the table. The tableware being used includes plates, cups and cutlery.

The robot system has three modules, the decision-maker module that includes the planner and the learner, the perception module that obtains a representation of the scene that can be used by the decision-maker, and the manipulation module that executes actions.

The perception module uses a camera that is located on top of the table (hanging from the ceiling) to update continuously a symbolic state that represents the table. This state consists of a set of literals that describe the different locations on the table. These literals are:

- “ArrivingPosition(?loc)” indicates that ?loc is a position where people will leave new tableware. This literal is constant as objects always appear on the bottom side of the table.

- “PickUpPosition(?loc)” indicates that ?loc is a position where mobile robots will pick up piles to take them to the kitchen. This literal is constant as mobile robots always pick up piles from the top side of the table.

- “mobileRobotPickingUp(?loc)” indicates that a mobile robot will pick up a pile during the next iteration.

- “plate(?loc)” indicates that there is at least one plate at ?loc.

- “cup(?loc)” indicates that there is at least one cup at ?loc.

- “cutlery(?loc)” indicates that there is at least one fork, knife or spoon at ?loc.

- “stable(?loc)” indicates that the pile at ?loc is stable. An unstable pile cannot be picked up by a mobile robot.

The manipulator module executes the actions planned. It takes the 3D perceptions obtained by the camera and generates the arm trajectories and gripper movements required to move the objects as desired. The symbolic actions that can be planned are:

- The “put(?loc1, ?loc2)” action to put one object from a pile in position ?loc1 into position ?loc2.

- The “movePile(?loc1, ?loc2)” action that drags a whole pile from ?loc1 to ?loc2 if ?loc2 is empty. This action has high failure rates with unstable piles.

Piles may become unstable if objects are not piled properly. For example, if a plate is placed on top of a pile containing a cup and a fork, there is a high probability that it will become unstable. To obtain stable piles, in general, plates should be placed at the bottom, cups on top of them, and cutlery at the top. Unstable piles are harder to move, and mobile robots cannot pick them successfully.

Finally the decision-maker module uses the V-MIN algorithm (Section 4.5) in combination with the learner presented in this chapter to learn, and the PROST planner (Keller and Eyerich, 2012) to plan.

Figures 5.9 and 5.10 show examples of the Table Clearing task. The robot plans the optimal action to execute based on the state of the table.

Learning a Model with RL

Using the setup described above, we executed the learner to obtain a model of the task from scratch while solving it. To make the experimentation easier, instead of having mobile robots picking up piles from the *pick up* area, a person did it. The robot had to solve experiments of increasing difficulty. All experiments lasted 30 iterations during which piles were picked up 6 times from the *pick up* area.

1. The easiest experiment was executed once. People left 10 objects on the table during the experiment, including only plates and cups.

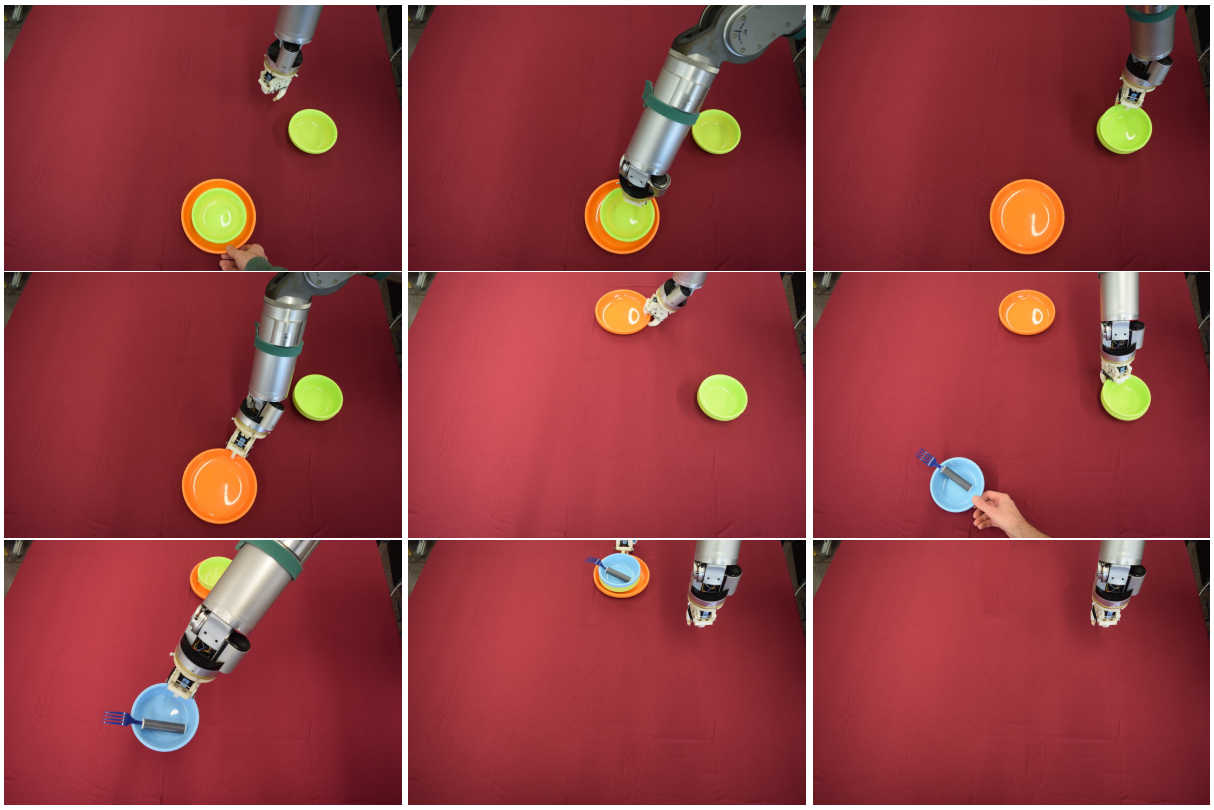


Figure 5.9: Table clearing with a learned model. As the robot does not know if new tableware will arrive shortly, it decides first to place the two cups together (images $1-3$), and then moves the plate to the *pick up* area (images $4-5$). Afterwards, a new pile arrives that contains no plates (image 6), so the robot decides to move the cups to the picking up area (image 7), and finally, as a mobile robot is arriving, the robot places the last pieces of tableware on the *pick up* area (image 8). Image 9 shows the final state after a mobile robot picked up the pile.

2. The medium experiment was executed twice. People left 13 objects on the table that included plates, cups and cutlery.

3. The difficult experiment was executed twice. People left 18 objects on the table that included plates, cups and cutlery.

The goals of the robot were to minimize the number of objects on the table, and to have empty space in the *arriving* area so that people always had space to place new tableware. Therefore big enough piles had to be given to the mobile robot at the same time that the arriving area had to be cleared quickly.

The learner parameters used were $\alpha = 0.01$, $\epsilon = 0.1$, $\omega = 2$, $\delta = 0.05$, $\kappa = 1000$, and the subsumption was enabled. The V-MIN exploration threshold was $\zeta = 3$ and V_{min} was selected and updated by the teacher depending on the robot performance.

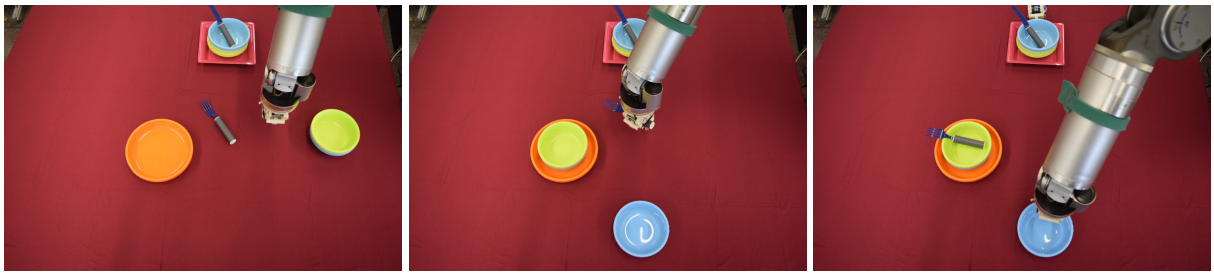


Figure 5.10: Table clearing with a learned model. **Initially**, the robot has already one pile prepared for the mobile robots. The planner opts to prepare a new pile with the tableware on the table as it expects that at some point a mobile robot will pick up the pile on top. **In the second image**, the new pile is halfway done, and a new cup arrives. The planner knows that in the next iteration a mobile robot is coming, so it decides to finish the new pile. **In the third image**, a mobile robot picks the top pile, and the manipulator robot plans that the best action is to move the new cup to the center free area to make more space in the arriving area. Once the mobile robot takes the top pile, the manipulator robot will plan the action of taking the center pile to the *pick up* area.

	Easy 1	Medium 1	Medium 2	Difficult 1	Difficult 2
Number of objects	10*	13	13	18	18
Sum of reward	-58	-95	-84	-157	-132
Optimal rewards	-41	-62	-65	-97	-101
Teacher requests	7	6	3	4	1

Table 5.1: Learning results in the robot table clearing task. The columns show the different learning experiments, which were executed in order of increasing difficulty. The rows show the number of objects that arrived in each experiment, the sum of rewards, and the number of teacher demonstration requests.
*In the easy setup no cutlery was used (only plates and cups).).

Learning Results

Table 5.1 summarizes the performance of the manipulator robot. During the easy experiment, 7 teacher demonstration requests were required to complete the task. The robot had to learn the "put" action to pick and place objects, and also the operators defining how mobile robots took piles from the *pick up* area.

In the second experiment, as cutlery appeared for the first time, and piles that have cutlery in the middle are usually unstable, the robot needed 6 extra demonstrations to learn how to manipulate cutlery and how unstable piles could be unpiled to make them stable again. The third experiment needed only 3 more demonstrations as the robot already knew most of the dynamics.

Finally, in the last difficult experiments, 5 teacher demonstrations were needed once many objects accumulated on the table. The robot had to learn that it could assemble a pile in the

	Optimal	Model with exogenous	Model with no exogenous
Sum of rewards	-99	-119	-136
Operators (actions)	-	8	20
Operators (exogenous)	-	11	0

Table 5.2: Execution results with the learned models. The columns show the results with an optimal sequence of actions, a learned model considering exogenous effects, and a learned model not considering exogenous effects. The rows show the sum of rewards and the number of planning operators that represent actions and exogenous effects.

middle area, and then move it directly to the *pick up* area by using the action "movePile(?loc1, ?loc2)". In previous experiments this action was not needed as less tableware arrived and the robot could just make piles directly on the *pick up* area with the "put" action.

It should be noted that during the learning experiments we observed that the effectiveness of the robot actions was very important. If actions failed often, many more action executions were needed to obtain a proper model. Therefore, the robot actions were improved so that they succeeded in most cases to make the task simpler.

Evaluation of the Learned Model

Finally, an experiment was carried out to evaluate the usefulness of learning exogenous effects. Here we compare the performance when using the model learned in the previous section, with another learned model that does not include exogenous effects. The exogenous effects allow the planner to know the probabilities with which each type of tableware arrives, and the probability with which mobile robots come to pick piles up. There was a reward discount of -1 for every object on the table at each step, and a discount of -5 if new tableware was arriving but the arriving area was full. The results are shown in Table 5.2.

- An optimal action sequence obtained a reward of -99. Note that this optimal action sequence was not obtained in a fair way: it was created knowing in advance when and which type of tableware was arriving at each step, and when a mobile robot approached. In contrast the manipulator robot did not have this information, it could only know the probabilities of these effects.

- The model containing exogenous effects obtained a reward of -119. In this experiment the manipulator robot performed well in general, but it created some inefficient piles as a mobile robot would take shorter or longer than expected to arrive. Moreover, the manipulator robot took a conservative strategy and it didn't complete fast enough some piles, as it did not know if new tableware would be arriving shortly. The model consisted of 8 planning operators for action effects, and 11 planning operators for exogenous effects.

- The model without exogenous effects obtained a reward of -136. This model had more trouble to complete the task. The manipulator robot did not know that new tableware was arriving, so it tried to make the best piles with the given objects, which resulted in the robot redoing the piles continuously. The model had 20 operators that represented action effects. As exogenous effects were not learned separately, each operator actually represented combinations of an action effect with exogenous effects. However, these operators were less effective to represent the model than separate action and exogenous operators, so the planner selected worse policies.

Analyzing the results we can see that the proposed learner allows a robot to learn tasks including exogenous effects. Given that only 5 episodes of 30 action executions were used for training, the results were good. The experiment also shows that a model considering exogenous effects improves the performance when external agents interact. Here the planner was able to find much better plans when it anticipated that new tableware or a mobile robot were arriving. However, extending this task with a richer representation and more external agents would be costly. For every literal and exogenous effect that gets added, the number of input experiences and computational time required to learn a model increases significantly.

Conclusions

We have introduced a new method that, given a set of input experiences, learns a general model explaining them. In contrast to previous approaches, it can learn exogenous effects (effects not related to any action), while still being similarly good at obtaining a relational representation of the problem and at learning uncertain effects. Moreover optimal and suboptimal search methods are provided, so the best approach can be chosen depending on the quality requirements, the difficulty of the problem, and the learning time available. The main limitation of the algorithm is scalability when the number of generated propositional atoms becomes large.

This learner also improved the behavior of RL algorithms, as the model learner was the limiting factor when tackling expressive models. We validated experimentally that the integration in the V-MIN algorithm allowed an agent to learn models with exogenous effects by executing a relatively small number of actions and teacher demonstration requests.

The learner was also integrated in a robot to perform the task of clearing the tableware on a table. In this task external agents interacted, people brought new tableware continuously and the manipulator robot had to cooperate with mobile robots to take the tableware to the kitchen. The learner was able to learn a usable model in just 5 episodes of 30 action executions. Finally, the model was used to complete the task, and learning models with such exogenous effects proved to increase the obtained reward significantly.

6
Conclusions

In this thesis we have worked on improving task performance with the use of automated planning. Our work combines learners to generate models of the task, planners to exploit those models, and RL algorithms to ensure that complete enough models are learned. We showed the benefits of those approaches in both simulated tasks and robotic tasks. To recapitulate, the contributions presented are:

1. We have shown that probabilistic planning can solve complex tasks faster than reactive strategies, and maintains a similar performance in easy tasks. We have also presented the importance of the trade-off between planning time and plan quality, which should be adapted to the complexity of each problem. Finally, we have shown that replanning every few actions is really helpful in very noisy tasks. All these results were proven experimentally in the task of cleaning dirty surfaces.

2. We have extended RL algorithms by adding the option of actively requesting demonstrations from a teacher to complete tasks faster. The agent can explore autonomously to learn the model, but it can also request teacher demonstrations to quickly overcome difficulties and learn new actions. The proposed methods learn with fewer action executions, adapt to changes in the tasks, and achieve the quality requirements of the user (where a better quality also implies more action executions and demonstration requests). Finally, we have explained how to analyze the learned models to extract the unlearned or problematic parts of the model. This information allow the agent to provide guidance to the teacher when a demonstration is requested, and to avoid irrecoverable errors.

3. We have presented a learner that obtains relational probabilistic models with action effects and exogenous effects that explain a set of input state transitions. This learner yields more expressive models than previous state-of-the-art learners, and as the learner was the limiting factor, it also allows the RL algorithms to solve more complex tasks.

All these works share the same context, that is, they are designed to minimize the action executions required to complete a task. Probabilistic planners directly obtain the best sequence of actions, while learners aim at obtaining the best model for the planner. Moreover, teacher demonstrations requests proved to be an excellent tool to get good models while minimizing exploration and action executions.

Finally, we evaluated our work in standard planning domains from the IPPC and robotic tasks. Our approaches could solve the tasks of clearing dirt on table, clearing the tableware on a table, and assembling industrial pieces successfully.

Additional Remarks

In this section we will cover some related lessons that we learned during the thesis.

First, even though most robotic applications have uncertainty, a probabilistic planner is not always the best way to solve them. Little and Thiebaux (2007a) provide a very good insight about the cases when a probabilistic planner should be used. If the task has only one goal trajectory and has no avoidable dead-ends, then a deterministic planner will be both faster and easier to implement. If something unexpected would happen, the cost of replanning with a deterministic planner would still be much lower than using a probabilistic planner. Note that most of the problems used in this thesis have avoidable dead-ends, so a probabilistic planner was needed, but there are many common robotic problems where a deterministic planner would suffice.

The main limitation of the techniques presented here is that planners and learners require a compact symbolic state representation. As the states are limited to binary literals, continuous variables have to be determinized, and the state-space can quickly become too large. When designing new domains, the number of literals should be minimized as much as possible to make the problems tractable.

Finally, it is important to put some effort in the human readability of the state predicates. Easily understandable predicates make the task much easier for engineers and teachers: planning excuses provide more information, and debugging unexpected states becomes easier.

Future Work

As we tackled different algorithms during the thesis, there are several directions for future work: improving the interaction with the teacher, learning more expressive models, and obtaining better planning excuses.

The interaction with the teacher is currently unidirectional, demonstrations are only executed when requested by the robot. However, in some applications it would be more natural if it was bidirectional, that is if the teacher could also actively provide suggestions to the robot when it did not perform as expected. Some approaches have used teacher suggestions to improve models (Walsh et al., 2010), but they require the teacher to continuously provide this feedback. Ideally, the teacher should be able to decide whether to monitor the robot and provide feedback, or to ignore the robot and provide help only when requested.

Another line of work is to extend model learners to tackle more expressive models. For example, existing planners support all the domains in the IPPC 2014 competition, but the learner presented in this thesis only supports a few of them. The main limitations is that universal and existential quantifiers are not supported. These quantifiers have been learned in deterministic domains (Zhuo et al., 2010), but it remains as future work to consider such quantifiers in probabilistic model learners.

Finally, several methods presented in this thesis, such as dead-end avoidance and teacher guidance, depend on the quality of planning excuses. The proposal by Göbelbecker et al. (2010) considers goal-driven deterministic domains, but more general methods could get better excuses in non-deterministic domains.

A
List of publications

In this section, the reader can find the complete list of accepted and submitted publications since the beginning of the PhD:

Journals

J1. D. MARTÍNEZ, G. ALENYÀ, T. RIBEIRO AND K. INOUE AND C. TORRAS. "Relational Reinforcement Learning for Planning with Exogenous Effects", submitted.

J2. D. MARTÍNEZ, G. ALENYÀ AND C. TORRAS. "Relational Reinforcement Learning with Guided Demonstrations", in *Artificial Intelligence*, 2016.

J3. D. MARTÍNEZ, G. ALENYÀ AND C. TORRAS. "Planning robot manipulation to clean planar surfaces", in *Engineering Applications of Artificial Intelligence*, 39: 23-32, 2015.

J4. T. R. SAVARIMUTHU, A. G. BUCH, C. SCHLETTE, N. WANTIA, J. ROSSMANN, D. MARTÍNEZ, G. ALENYÀ, C. TORRAS, A. UDE, B. NEMEC, A. KRAMBERGER, F. WÖRGÖTTER, E. E. AKSOY, J. PAPON, S. HALLER, J. PIATER AND N. KRÜGER. "Teaching a Robot the Semantics of Assembly Tasks", in *IEEE Transactions on Systems, Man and Cybernetics: Systems*, accepted with minor revision.

Conferences

C1. D. MARTÍNEZ, G. ALENYÀ, C. TORRAS, T. RIBEIRO AND K. INOUE. "Learning Relational Dynamics of Stochastic Domains for Planning", in *proceedings of the International Conference on Automated Planning and Scheduling*, 2016.

C2. D. MARTÍNEZ, G. ALENYÀ AND C. TORRAS. "Safe robot execution in model-based reinforcement learning", in *proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015, pp. 6422-6427.

C3. D. MARTÍNEZ, G. ALENYÀ AND C. TORRAS. "V-MIN: Efficient reinforcement learning through demonstrations and relaxed reward demands", in *proceedings of the AAAI Conference on Artificial Intelligence*, 2015, pp. 2857–2863

C4. D. MARTÍNEZ, G. ALENYÀ, P. JIMÉNEZ, C. TORRAS, J. ROSSMANN, N. WANTIA, A. EREN ERDAL, S. HALLER AND J. PIATER. "Active learning of manipulation sequences", in *proceedings of the IEEE International Conference on Robotics and Automation*, 2014, Hong Kong, pp. 5671-5678.

Workshops

W1. D. MARTÍNEZ, T. RIBEIRO, K. INOUE, G. ALENYÀ AND C. TORRAS. "Learning probabilistic action models from interpretation transitions", in *Technical Communications of the International Conference on Logic Programming*, 2015, Vol 1433 of CEUR Workshop Proceedings.

W2. D. MARTÍNEZ, G. ALENYÀ AND C. TORRAS. "Finding Safe Policies in Model-Based Active Learning", in *proceedings of the IROS Machine Learning in Planning and Control of Robot Motion Workshop*, 2014, pp. 1-6.

W3. T. R. SAVARIMUTHU, A. G. BUCH, Y. YANG, S. HALLER, J. PAPON, D. MARTÍNEZ AND E. E. AKSOY. "Manipulation Monitoring and Robot Intervention in Complex Manipulation Sequences", in *proceedings of the RSS Workshop on Robotic Monitoring*, 2014.

W4. D. MARTÍNEZ, G. ALENYÀ AND C. TORRAS. "Planning surface cleaning tasks by learning uncertain drag actions outcomes", in *proceedings of the ICAPS Workshop on Planning and Robotics*, 2013, Rome, pp. 106-111.

Bibliography

A. Agostini, C. Torras, and F. Wörgötter. Efficient interactive decision-making framework for robotic applications. *Artificial Intelligence*, to appear, 2016.

E. E. Aksoy, A. Abramov, J. Dörr, K. Ning, B. Dellen, and F. Wörgötter. Learning the semantics of object–action relations by observation. *International Journal of Robotics Research*, 30(10): 1229–1249, 2011.

B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

R. A. Bianchi, L. A. Celiberto, P. E. Santos, J. P. Matsuura, and R. L. de Màntaras. Transferring knowledge as heuristics in reinforcement learning: A case-based approach. *Artificial Intelligence*, 226:102–121, 2015.

B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003.

R. Bormann, F. Weisshardt, G. Arbeiter, and J. Fischer. Autonomous dirt detection for cleaning in office environments. In *Proc. of Int. Conf. on Robotics and Automation*, pages 1260–1267, 2013.

R. I. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2003.

A. G. Buch, D. Kraft, J.-K. Kamarainen, H. G. Petersen, and N. Kruger. Pose estimation using local structure-specific shape and appearance context. In *Proc. of International Conference on Robotics and Automation*, pages 2080–2087, 2013.

S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *Int. Journal of Robotic Research*, 28(1):104–126, 2009.

S. Chernova and M. Veloso. Interactive policy learning through confidence-based autonomy. *Journal of Artificial Intelligence Research*, 34(1):1–25, 2009.

A. Cocora, K. Kersting, C. Plagemann, W. Burgard, and L. De Raedt. Learning relational navigation policies. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 2792–2797, 2006.

A. Coles, A. Coles, A. G. Olaya, S. Jiménez, C. L. López, S. Sanner, and S. Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1):83–88, 2012.

A. Deshpande, B. Milch, L. S. Zettlemoyer, and L. P. Kaelbling. Learning probabilistic relational dynamics for multiple tasks. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 83–92, 2007.

C. Diuk, A. Cohen, and M. L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the International Conference on Machine learning*, pages 240–247, 2008.

S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001.

A. Gams, M. Do, A. Ude, T. Asfour, and R. Dillmann. On-line periodic movement and force-profile learning for adaptation to new surfaces. In *Proc. of Int. Conf. on Humanoid Robots*, pages 560–565, 2010.

M. Göbelbecker, T. Keller, P. Eyerich, M. Brenner, and B. Nebel. Coming up with good excuses: What to do when no plan can be found. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 81–88, 2010.

D. H. Grollman and O. C. Jenkins. Dogged learning for robots. In *Proceedings of the International Conference on Robotics and Automation*, pages 2483–2488, 2007.

M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246, 2006.

J. Hess, J. Sturm, and W. Burgard. Learning the state transition model to efficiently clean surfaces with mobile manipulation robots. In *Proc. of ICRA Workshop on Manipulation under Uncertainty*, 2011.

J. Hess, M. Beinhofer, D. Kuhner, P. Ruchti, and W. Burgard. Poisson-driven dirt maps for efficient robot cleaning. In *Proc. of Int. Conf. on Robotics and Automation*, pages 2245–2250, 2013.

T. Hester and P. Stone. Texplore: real-time sample-efficient reinforcement learning for robots. *Machine learning*, 90(3):385–429, 2013.

J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

K. Inoue, T. Ribeiro, and C. Sakama. Learning from interpretation transition. *Machine Learning*, 94(1):51–79, 2014.

S. Jiménez, F. Fernández, and D. Borrajo. The PELA architecture: integrating planning and learning to improve execution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1294–1299, 2008.

L. P. Kaelbling and T. Lozano-Pérez. Unifying perception, estimation and action for mobile manipulation via belief space planning. In *Proc. of Int. Conf. on Robotics and Automation*, pages 2952–2959, 2012.

D. Katz, Y. Pyuro, and O. Brock. Learning to manipulate articulated objects in unstructured environments using a grounded relational representation. In *Proceedings of Robotics: Science and Systems*, pages 254–261, 2008.

M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002.

T. Keller and P. Eyerich. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 119–127, June 2012.

L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML*, pages 282–293, 2006.

A. Kolobov, P. Dai, Mausam, and D. S. Weld. Reverse iterative deepening for finite-horizon MDPs with large branching factors. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 146–154, 2012a.

A. Kolobov, Mausam, and D. S. Weld. LRTDP versus UCT for online probabilistic planning. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2012b.

P. Kormushev, D. N. Nenchev, S. Calinon, and D. G. Caldwell. Upper-body kinesthetic teaching of a free-standing humanoid robot. In *Proc. of Int. Conf. on Robotics and Automation*, pages 3970–3975, 2011.

N. Krüger, C. Geib, J. Piater, R. Petrick, M. Steedman, F. Wörgötter, A. Ude, T. Asfour, D. Kraft, D. Omrcen, et al. Object-action complexes: Grounded abstractions of sensory-motor processes. *Robotics and Autonomous Systems*, 59(10):740–757, 2011.

T. Lang and M. Toussaint. Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research*, 39:1–49, 2010.

T. Lang, M. Toussaint, and K. Kersting. Exploration in relational domains for model-based reinforcement learning. *Journal of Machine Learning Research*, 13:3691–3734, 2012.

L. Li. *A unifying framework for computational reinforcement learning theory*. PhD thesis, Rutgers, The State University of New Jersey, 2009.

L. Li, M. L. Littman, T. J. Walsh, and A. L. Strehl. Knows what it knows: a framework for self-aware learning. *Machine learning*, 82(3):399–443, 2011.

N. Lipovetzky, M. Ramirez, and H. Geffner. Classical planning with simulators: Results on the atari video games. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1610–1616, 2015.

I. Little and S. Thiebaux. Probabilistic planning vs. replanning. In *Proc. of ICAPS Workshop on IPC: Past, Present and Future*, 2007a.

I. Little and S. Thiebaux. Probabilistic planning vs. replanning. In *Proceedings of the ICAPS Workshop on IPC: Past, Present and Future*, 2007b.

M. L. Littman. Reinforcement learning improves behaviour from evaluative feedback. *Nature*, 521(7553):445–451, 2015.

D. Martínez, G. Alenyà, and C. Torras. Planning surface cleaning tasks by learning uncertain drag actions outcomes. In *ICAPS Workshop on Planning and Robotics (PlanRob)*, pages 106–111, 2013.

D. Martínez, G. Alenyà, P. Jiménez, C. Torras, J. Rossmann, N. Wantia, E. E. Aksoy, S. Haller, and J. Piater. Active learning of manipulation sequences. In *2014 IEEE International Conference on Robotics and Automation*, pages 5671–5678, 2014a.

D. Martínez, G. Alenyà, and C. Torras. Finding safe policies in model-based active learning. In *IROS workshop on Machine Learning in Planning and Control of Robot Motion*, 2014b.

D. Martínez, G. Alenyà, and C. Torras. Planning robot manipulation to clean planar surfaces. *Engineering Applications of Artificial Intelligence*, 39(March 2015):23–32, 2015a.

D. Martínez, G. Alenyà, and C. Torras. Safe robot execution in model-based reinforcement learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 6422–6427, 2015b.

D. Martínez, G. Alenyà, and C. Torras. V-MIN: Efficient reinforcement learning through demonstrations and relaxed reward demands. In *The Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*, pages 2857–2863, 2015c.

D. Martínez, T. Ribeiro, K. Inoue, G. Alenyà, and C. Torras. Learning probabilistic action models from interpretation transitions. *Technical Communication of the International Conference on Logic Programming, CEUR Workshop Proceedings*, 1433(30), 2015d.

D. Martínez, G. Alenyà, and C. Torras. Relational reinforcement learning with guided demonstrations. *Artificial Intelligence*, 2016a.

D. Martínez, G. Alenyà, C. Torras, T. Ribeiro, and K. Inoue. Learning relational dynamics of stochastic domains for planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 235–243, 2016b.

D. Martínez, G. Alenyà, T. Ribeiro, K. Inoue, and C. Torras. Relational reinforcement learning for planning with exogenous effects. *submitted*, 2017.

M. V. Menezes, L. N. de Barros, and S. do Lago Pereira. Planning task validation. *Proceedings of the Scheduling and Planning Applications Workshops in ICAPS*, pages 48–55, 2012.

Ç. Meriçli, M. Veloso, and H. L. Akın. Multi-resolution corrective demonstration for efficient task execution and refinement. *International Journal of Social Robotics*, 4(4):423–435, 2012.

B. Moldovan, P. Moreno, M. van Otterlo, J. Santos-Victor, and L. De Raedt. Learning relational affordance models for robots in multi-object manipulation tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 4373–4378, 2012.

K. Mourão. Learning probabilistic planning operators from noisy observations. In *Proceedings of the Workshop of the UK Planning and Scheduling Special Interest Group*, 2014.

K. Mourão, L. S. Zettlemoyer, R. Petrick, and M. Steedman. Learning strips operators from noisy and incomplete observations. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 614–623, 2012.

B. Nemec and A. Ude. Action sequencing using dynamic movement primitives. *Robotica*, 30(5): 837, 2012.

J. Papon, T. Kulvicius, E. E. Aksoy, and F. Wörgötter. Point cloud video object segmentation using a persistent supervoxel world-model. In *Proc. of International Conference on Intelligent Robots and Systems*, 2013.

H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29(1):309–352, 2007.

G. Randelli, T. M. Bonanni, L. Iocchi, and D. Nardi. Knowledge acquisition through human–robot multimodal interaction. *Intelligent Service Robotics*, 6(1):19–31, 2013.

T. Ribeiro and K. Inoue. Learning prime implicant conditions from interpretation transition. In *Proceedings of the International Conference on Inductive Logic Programming*, 2014.

C. Rodrigues, P. Gérard, and C. Rouveirol. Incremental learning of relational action models in noisy environments. In *Proceedings of the International Conference on Inductive Logic Programming*, pages 206–213, 2011.

J. Rossmann, C. Schlette, and N. Wantia. Virtual reality in the loop - providing an interface for an intelligent rule learning and planning system. In *ICRA workshop in Semantics, Identification and Control of Robot-Human-Environment Interaction*, pages 60–65, 2013.

S. Sanner. Relational dynamic influence diagram language (RDDL): Language description. *Unpublished ms. Australian National University*, 2010.

F. Sato, T. Nishii, J. Takahashi, Y. Yoshida, M. Mitsuhashi, and D. Nenchev. Experimental evaluation of a trajectory/force tracking controller for a humanoid robot cleaning a vertical surface. In *Proc. of Int. Conf. on Intelligent Robots and Systems*, pages 3179–3184, 2011.

T. R. Savarimuthu, D. Liljekrans, L.-P. Ellekilde, A. Ude, B. Nemec, and N. Kruger. Analysis of human peg-in-hole executions in a robotic embodiment using uncertain grasps. In *Proc. of International Workshop on Robot Motion and Control*, pages 233–239, 2013.

T. R. Savarimuthu, A. G. Buch, Y. Yang, W. Mustafar, S. Haller, J. Papon, D. Martínez, and E. E. Aksoy. Manipulation monitoring and robot intervention in complex manipulation sequences. In *RSS Workshop on Robotic Monitoring*, 2014.

T. R. Savarimuthu, A. G. Buch, C. Schlette, N. Wantia, J. Roß mann, D. Martínez, G. Alenyà, C. Torras, A. Ude, B. Nemec, A. Kramberger, F. Wörgötter, E. E. Aksoy, J. Papon, S. Haller, J. Piater, and N. Krüger. Teaching a robot the semantics of assembly tasks. *IEEE Transactions on Systems, Man and Cybernetics: Systems (accepted with minor revision)*, 2016.

A. L. Strehl, L. Li, and M. L. Littman. Reinforcement learning in finite mdps: Pac analysis. *Journal of Machine Learning Research*, 10:2413–2444, 2009.

D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue. Learning revised models for planning in adaptive systems. In *Proceedings of the International Conference on Software Engineering*, pages 63–71, 2013.

M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685, 2009.

I. Thon, N. Landwehr, and L. De Raedt. Stochastic relational processes: Efficient inference and applications. *Machine Learning*, 82(2):239–272, 2011.

M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, and S. Sanner. The 2014 international planning competition: Progress and trends. *AI Magazine*, 36(3):90–98, 2015.

T. S. Vaquero, J. R. Silva, and J. C. Beck. Post-design analysis for building and refining ai planning systems. *Engineering Applications of Artificial Intelligence*, 26(8):1967–1979, 2013.

T. J. Walsh. *Efficient learning of relational models for sequential decision making*. PhD thesis, Rutgers, The State University of New Jersey, 2010.

T. J. Walsh and M. L. Littman. Efficient learning of action schemas and web-service descriptions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 714–719, 2008.

T. J. Walsh, I. Szita, C. Diuk, and M. L. Littman. Exploring compact reinforcement-learning representations with linear regression. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 591–598, 2009.

T. J. Walsh, K. Subramanian, M. L. Littman, and C. Diuk. Generalizing apprenticeship learning across hypothesis classes. In *Proceedings of the International Conference on Machine Learning*, pages 1119–1126, 2010.

S. W. Yoon, A. Fern, and R. Givan. FF-replan: A baseline for probabilistic planning. In *Proc. of Int. Conf. on Automated Planning and Scheduling*, volume 7, pages 352–359, 2007.

H. L. Younes and M. L. Littman. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Technical Report CMU-CS-04-162*, 2004.

H. L. S. Younes, M. L. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research*, 24:851–887, 2005.

H. H. Zhuo and S. Kambhampati. Action-model acquisition from noisy plan traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 2444–2450, 2013.

H. H. Zhuo and Q. Yang. Action-model acquisition for planning via transfer learning. *Artificial intelligence*, 212:80–103, 2014.

H. H. Zhuo, Q. Yang, D. H. Hu, and L. Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540–1569, 2010.