Planning in Action Language $\mathcal{BC}$ while Learning Action Costs for Mobile Robots

Piyush Khandelwal, Fangkai Yang, Matteo Leonetti, Vladimir Lifschitz and Peter Stone
Department of Computer Science
The University of Texas at Austin
2317 Speedway, Stop D9500
Austin, TX 78712, USA
{piyushk,fkyang,matteo,vl,pstone}@cs.utexas.edu

Abstract

The action language $\mathcal{BC}$ provides an elegant way of formalizing dynamic domains which involve indirect effects of actions and recursively defined fluents. In complex robot task planning domains, it may be necessary for robots to plan with incomplete information, and reason about indirect or recursive action effects. In this paper, we demonstrate how $\mathcal{BC}$ can be used for robot task planning to solve these issues. Additionally, action costs are incorporated with planning to produce optimal plans, and we estimate these costs from experience making planning adaptive. This paper presents the first application of $\mathcal{BC}$ on a real robot in a realistic domain, which involves human-robot interaction for knowledge acquisition, optimal plan generation to minimize navigation time, and learning for adaptive planning.

Introduction

As robots deal with increasingly complex tasks, automated planning systems can provide great flexibility over direct implementation of behaviors. In mobile robotics, uncertainty about the environment stems from many sources, which is particularly true for domains inhabited by humans, where the state of the environment can change outside the robot's control in ways that are difficult to predict. The qualitative modeling of dynamic domains at a given abstraction level, based on a formal language, allows for the generation of provably correct plans. The brittleness owing to the prevalent uncertainty in the model can be overcome through execution monitoring and replanning, when the outcome of an action deviates from the expected effect.

Action languages are attractive in robotic domains for the reason that they solve the *frame problem* (McCarthy and Hayes 1969), solve the *ramification problem* (Finger 1986) by formalizing indirect effects of actions, and are *elaboration tolerant* (McCarthy 1987). Existing tools such as COALA (Gebser, Grote, and Schaub 2010) and CPLUS2ASP (Babb and Lee 2013) allow us to translate action descriptions into logic programs under answer set semantics (Gelfond and Lifschitz 1988; 1991), and planning can be accomplished using computational methods of Answer Set Programming (ASP) (Marek and Truszczynski 1999; Niemelä 1999). Furthermore, the action language $\mathcal{BC}$ (Lee, Lifschitz, and Yang 2013) can easily formalize recursively defined fluents, which can be useful in robot task planning.

The main contribution of this paper is a demonstration that the action language $\mathcal{BC}$ can be used for robot task planning in realistic domains, that require planning in the presence of missing information and indirect action effects. These features are necessary to completely describe many complex tasks. For instance, in a task where a robot has to collect mail intended for delivery from all building residents, the robot may need to visit a person whose location it does not know. To overcome this problem, it can plan to complete its task by asking someone else for that person's location, thereby acquiring this missing information. Additionally, a person may forward his mail to another person in case he will be unavailable when the robot comes around to collect mail. In such situations, the information about mail transfers is best expressed through a recursive definition. When the robot visits a person who has mail from multiple people, planning needs to account for the fact that mail from all these people will be collected indirectly. In this paper, we use this mail collection task to demonstrate how these problems are solved. The overall methodology is applicable to other planning domains that involve recursive fluents, indirect action effects, and human-robot interaction.

The second contribution of this paper is to show how answer set planning under action costs (Eiter et al. 2003) can be applied to robot task planning, and how these costs can be learned from experience. Incorporating costs in symbolic planning is important for applications that involve physical systems and deal with limited resources such as time, battery, communication bandwidth, etc. Previous applications of action languages for robotics do not consider these costs (Caldiran et al. 2009; Chen et al. 2010). It is also important to learn costs from the environment, since these costs may not be the same for different robots, and may even differ for the same robot under different environmental conditions. For instance, while a fully articulated humanoid robot may be slower than a wheeled robot for navigation tasks, the extra dexterity it possesses may allow it to be faster at opening doors. Similarly, construction inside a building may render certain paths slow to navigate. If the robot learns these costs on the fly, it becomes unnecessary to worry about them during the domain formalization.

We evaluate our approach using a Segway mobile robot navigating through an indoor environment and interacting with people, serving people by completing tasks such as collecting mail. We also demonstrate the process of learning navigation costs both in a simulation environment and on a physical robot. All the code used in this paper has been implemented using the ROS middleware package (Quigley et al. 2009) and the GAZEBO simulator (Koenig and Howard 2004), and is available in the public domain [1].

Related Work

Task planning problems for mobile robots can be described in the Planning Domain Definition Language (PDDL) (Quintero et al. 2011b), which can then be solved by general purpose PDDL planners such as SAYPHI (de la Rosa, Olaya, and Borrajo 2007), However, PDDL planning is mainly limited to domains in which all effects of actions are described directly, without specifying interactions between fluents. It should be noted that the original specification of PDDL includes axioms, which can specify indirect effects of actions. This feature, however, is rarely used in the planning community or planning competitions[2].

Answer set programming provides a clear semantics for indirect effects of actions. Indirect effects of actions can also be described in action languages such as $\mathcal{B}$ (Gelfond and Lifschitz 1998), $\mathcal{C}$ (McCain and Turner 1997), $\mathcal{C}+$ (Giunchiglia et al. 2004) and the recently proposed $\mathcal{BC}$ (Lee, Lifschitz, and Yang 2013), which can also support recursive action effects. Answer set programming has been successfully used to model the reactive control system of the space shuttle (Balduccini, Gelfond, and Nogueira 2006), and the action language $\mathcal{C}+$ has been used for robot task planning (Caldiran et al. 2009; Chen et al. 2010; Chen, Jin, and Yang 2012; Erdem and Patoglu 2012; Erdem et al. 2013; Havur et al. 2013). In most of these robotics applications, complete information for the initial state is available. In contrast, we are interested in large and realistic domains that require planning with incomplete information.

Recent work improves on existing ASP approaches for robot task planning by both using larger domains in simulation, as well as incorporating a constraint on the total time required to complete the goal (Erdem, Aker, and Patoglu 2012). While this previous work attempts to find the shortest plan that satisfies the goal within a prespecified time constraint, our work attempts to explicitly minimize the overall cost to produce the optimal plan. Additionally, this previous work attempts to include geometric reasoning at the planning level, and the ASP solver considers a discretized version of the true physical location of the robot. Since we target larger domains, we use a coarser discretization of the robot's location to keep planning scalable and use dedicated low-level control modules to navigate the robot.

The difficulty in modeling the environment has motivated a number of different combinations of planning and learning methods. In related work, a breadth-first planner is used to compute the set of shortest strong solutions in a non-deterministic PDDL domain, and the resulting plans are compiled into a state machine (Leonetti, Iocchi, and Patrizi 2012). Such a state machine is used at run time to constrain the agent's behavior, and learn the optimal plan through model-free reinforcement learning. While model-free learning has its benefits, the learned information cannot be reused on different tasks. Since the main reason to have a planner is often the need for such flexibility, in this paper we let the agent learn the cost of single actions, adapting the model to the actual environment.

The approach most closely related to ours, for learning individual action costs, is the PELA architecture (Jiménez, Fernández, and Borrajo 2013). In PELA, a PDDL description of the domain is augmented with cost information learned in a relational decision tree (Blockeel and De Raedt 1998). The cost computed for each action is such that the planner minimizes the probability of plan failures in their system. Our method estimates costs more generally based on any metric observable by the robot. Some preliminary work has been done in PELA to learn expected action durations (Quintero et al. 2011a), using a variant of relational decision trees. In contrast, we learn costs using exponentially weighted averaging, which allows us to respond to recent changes in the environment.

In preliminary work, we explored optimal planning and cost learning on robots using ASP (Yang et al. 2014). In this paper, we use the action language $\mathcal{BC}$ for planning, explore tasks where recursive fluents are necessary, and demonstrate our approach on real robots.

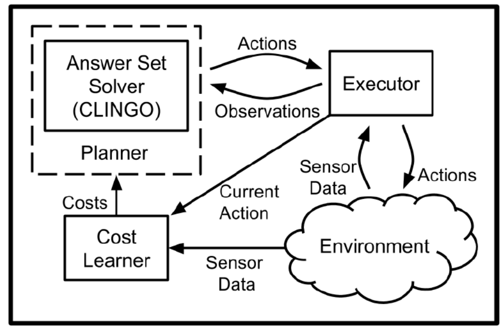


Figure 1: The architecture used in our approach. The planner invokes a cost learner that learns costs from sensing during execution.

Architecture Description

Our proposed architecture (shown in Figure 1) has two modules that constitute the decision making: a planning module, and a cost estimation module. At planning time, the planner generates a description of the initial state of the world based on observations provided by the executor. The initial state, domain description (translated from the action language $\mathcal{BC}$ into ASP), and goal description are sent to an answer set solver, in our case CLINGO (Gebser et al. 2011). CLINGO polls the cost of each action from the estimator, and produces an optimal plan. After plan generation, the executor invokes the appropriate controllers for each action, grounds

[1] `https://github.com/utexas-bwi/bwi_planning`

[2] The use of axioms is known to increase the expressiveness and elegance of the problem representation, and improve the performance of the planner (Thiébaux, Hoffmann, and Nebel 2003).

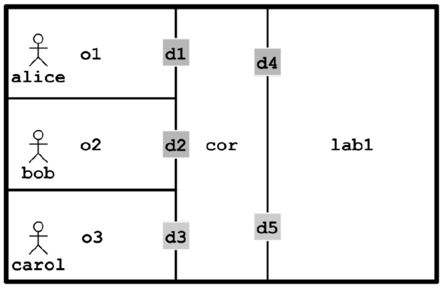


Figure 2: The layout of the example floor plan used in the text along with the rigid knowledge provided to the planner.

numeric sensor observations into symbolic fluents, and returns these fluents to the planning module for verification. If the observed fluents are incompatible with the state expected by the planner, the planner will update the robot's domain knowledge and replan. During action execution, the cost estimator also receives sensor data, and employs a learning algorithm to estimate the value of the cost of each action from the experienced samples. This architecture treats the planning module as a black box, and can in principle be adopted with any metric planner. Additionally, the formalism used to represent the state space in the planner and the cost estimator modules need not to be the same.

Domain Representation

Background: The Action Language $\mathcal{BC}$

The action language $\mathcal{BC}$, like other action description languages, describes dynamic domains as transition systems. An action description in the language $\mathcal{BC}$ includes two kinds of finite symbol sets, *fluent constants* and *action constants*. Fluent constants are further divided into *regular* and *statically determined*. Informally, regular fluents are those that are directly affected by actions, while statically determined fluents are those that are determined by other fluents. Every fluent constant has a finite *domain* of cardinality ≥ 2.

An *atom* is an expression of the form $f = v$, where f is a fluent constant, and v is an element of its domain. If the domain of f is $\{\mathbf{f}, \mathbf{t}\}$ then we say that f is *Boolean*. If f is Boolean then we will write the atom $f = \mathbf{t}$ as f, and the atom $f = \mathbf{f}$ as $\sim f$.

A *static law* is an expression of the form:

$$A_0 \textbf{ if } A_1, \ldots, A_m \textbf{ ifcons } A_{m+1}, \ldots, A_n$$

where $n \geq m \geq 0$ and each A_i is an atom. It expresses, informally speaking, that every state satisfies A_0 if it satisfies $A_1, \ldots, A_m$, and it can be assumed without contradiction that the state satisfies $A_{m+1}, \ldots, A_n$. If $m = 0$, then **if** is dropped; if $m = n$, then **ifcons** is dropped.

A *dynamic law* is an expression of the form:

$$A_0 \textbf{ after } A_1, \ldots, A_m \textbf{ ifcons } A_{m+1}, \ldots, A_n$$

where:

- $n \geq m \geq 0$,
- A_0 is an atom containing a regular fluent constant,
- $A_1, \ldots, A_m$ are atoms or action constants, and
- $A_{m+1}, \ldots, A_n$ are atoms.

It expresses, informally speaking, that the end state of any transition satisfies A_0 if its beginning state and its action satisfy $A_1, \ldots, A_m$, and it can be assumed without contradiction that the end state satisfies $A_{m+1}, \ldots, A_n$. If $m = n$, then **ifcons** is dropped.

An *action description* in the language $\mathcal{BC}$ is a finite set consisting of static and dynamic laws.

The following abbreviations are usually used in action descriptions. Symbols $a, a_1, \ldots, a_k$ denote action constants, $A, A_0, \ldots, A_m$ denote atoms, and f denotes a fluent.

abbreviation	*causal laws*
a **causes** A	A **after** a
a **causes** A_0 **if** $A_1, \ldots, A_m$	A_0 **after** $a, A_1, \ldots, A_m$
default A_0	A_0 **ifcons** A_0
inertial f	$f = v$ **after** $f = v$ **ifcons** $f = v$, for all v in the domain of f
nonexecutable $a_1, \ldots, a_k$ **if** $A_1, \ldots, A_m$,	$f = v$ **after** $a_1, \ldots, a_k, A_1, \ldots, A_m$ $f = w$ **after** $a_1, \ldots, a_k, A_1, \ldots, A_m$ $(v \neq w)$

The semantics of action descriptions in $\mathcal{BC}$ are defined by translation into the language of logic programs under answer set semantics. Automated translation of $\mathcal{BC}$ action descriptions is incorporated in software CPLUS2ASP version 2 (Babb and Lee 2013). Therefore, planning with $\mathcal{BC}$ can be automated by translating into answer set programming and calling answer set solvers.

Formalizing the Dynamic Domain

In order to demonstrate how ASP can be used for robot task planning under incomplete information, with human-robot interaction and with action costs, we use a small domain as a running example. The example domain we consider has a mobile robot that navigates inside a building, visiting and serving the inhabitants by collecting mail:

> The robot drops by offices at 2pm every day to collect outgoing mail from the residents. However, some people may not be in their offices at that time, so they can pass their outgoing mail to colleagues in other offices, and send this information to the robot. When the robot collects the mail, it should obtain it while only visiting people as necessary. If the robot needs to collect mail from a person whose location is not known, it should plan to visit other people to acquire this information.

We will show that solving this problem requires handling indirect effects of actions formulated by recursively defined fluents, which cannot be easily handled by planners based on the PDDL formulation, nor by previous action languages such as $\mathcal{C}$ and $\mathcal{C}+$. Solving it also involves planning under incomplete information and knowledge acquisition through human-robot interaction, which is not explored in existing work that uses action languages for task planning.

The floor plan of the example building is illustrated in Figure 2, along with information about the residents. In the experimental section, we evaluate our approach on a larger domain based on a real building.

In this example, we consider the following objects:

- *alice*, *bob*, *carol* and *dan* are people
- o_1, o_2, o_3 are offices and lab_1 is a lab.
- *cor* (corridor) is a room.
- d_1, d_2, d_3, d_4 and d_5 are doors.

In the following subsections, we will use meta-variables $P, P_1, P_2, \ldots$ to denote people, $R, R_1, R_2, \ldots$ to denote rooms, offices and labs, and $D, D_1, D_2, \ldots$ to denote doors.

Domain knowledge about a building includes the following three different types of information: *rigid knowledge*, *time-dependent knowledge*, and *action knowledge*. We explain each of these in detail in the following subsections.

Rigid Knowledge includes information about the building that does not depend upon the passage of time. In our example, rigid knowledge includes accessibility between the rooms, the lab, and the corridor. This knowledge has been formalized in our system as follows:

- $\mathit{hasdoor}(R, D)$: office R has door D. An office R does not have a door D unless specified. This default expresses the *closed world assumption* (Reiter 1978) for *hasdoor*:

$$\begin{array}{l} \mathit{hasdoor}(o_1, d_1) \quad \mathit{hasdoor}(o_2, d_2) \quad \mathit{hasdoor}(o_3, d_3) \\ \mathit{hasdoor}(lab_1, d_4) \quad \mathit{hasdoor}(lab_1, d_5) \\ \textbf{default} \ \sim\!\mathit{hasdoor}(R, D). \end{array}$$

- $\mathit{acc}(R_1, D, R_2)$: room R_1 is accessible from room R_2 via door D. Two rooms are not connected by a door unless specified:

$$\begin{array}{l} \mathit{acc}(R, D, \mathit{cor}) \ \textbf{if} \ \mathit{hasdoor}(R, D) \\ \mathit{acc}(R, D, \mathit{cor}) \ \textbf{if} \ \mathit{acc}(\mathit{cor}, D, R) \\ \textbf{default} \ \sim\!\mathit{acc}(R_1, D, R_2). \end{array}$$

- $\mathit{knows}(P_1, P_2)$ describes P_1 knows where person P_2 is. By default, a person P_1 does not know where another person P_2 is.
- $\mathit{passto}(P_1, P_2)$: person P_1 has passed mail to person P_2. By default, a person P_1 has not passed mail to a person P_2 (including himself).

Time-Dependent Knowledge includes information about the environment that can change with the passage of time, as the robot moves around in the environment. Time-dependent knowledge can be formalized as follows:

- The current location of a person is formalized by the fluent *inside*. $\mathit{inside}(P, R)$ means that person P is located in room R. A person can only be inside a single room at any given time. The fluent is *inertial*[3]:

$$\begin{array}{l} \sim\!\mathit{inside}(P, R_2) \ \textbf{if} \ \mathit{inside}(P, R_1) \qquad (R_1 \neq R_2) \\ \textbf{inertial} \ \mathit{inside}(P, R). \end{array}$$

- Whether the robot knows the current location of a person is formalized by the fluent *knowinside*. $\mathit{knowinside}(P, R)$ means the robot knows that person P is located in room R. The robot knows that a person can only be inside a single room at any given time. The fluent is inertial:

$$\begin{array}{l} \sim\!\mathit{knowinside}(P, R_2) \ \textbf{if} \ \mathit{knowinside}(P, R_1) \quad (R1 \neq R_2) \\ \textbf{inertial} \ \mathit{knowinside}(P, R). \end{array}$$

  If the robot knows that a person P is in room R, then P is indeed in room R:

$$\mathit{inside}(P, R) \ \textbf{if} \ \mathit{knowinside}(P, R).$$

- $\mathit{open}(D)$: a door D is open. By default, a door is not open.
- $\mathit{visiting}(P)$: the robot is visiting a person P. By default, a robot is not visiting anyone.
- $\mathit{mailcollected}(P)$: the robot has collected mail from P. This fluent is inertial. It is recursively defined as follows. The robot has collected P_1's mail if it has collected P_2's mail and P_1 has passed his mail to P_2.

$$\mathit{mailcollected}(P_1) \ \textbf{if} \ \mathit{mailcollected}(P_2), \mathit{passto}(P_1, P_2). \quad (1)$$

  Defining fluents recursively is a feature of the domain that cannot be easily formalized and planned using PDDL, but can be easily formalized in $\mathcal{BC}$.
- $\mathit{facing}(D)$: the robot is next to a door D and is facing it. The robot cannot face two different doors simultaneously.
- $\mathit{beside}(D)$: the robot is next to door D. $\mathit{beside}(D)$ is true if *facing(D)* is true, and the robot cannot be beside two different doors simultaneously. Since *beside* is implied by *facing*, it will become an indirect effect of the actions that make the fluent *facing* true.
- $\mathit{loc} = R$: the robot is at room R.

Action knowledge includes the rules that formalize the actions of the robot, the preconditions for executing those actions, and the effects of those actions. The robot can execute the following actions:

- $\mathit{approach}(D)$: the robot approaches door D. The robot can only approach a door accessible from the the robot's current location and if it is not facing the door. Approaching a door causes the robot to face that door.

$$\begin{array}{l} \mathit{approach}(D) \ \textbf{causes} \ \mathit{facing}(D) \\ \textbf{nonexecutable} \ \mathit{approach}(D) \ \textbf{if} \ \mathit{loc} = R, \sim\!\mathit{hasdoor}(R, D) \\ \textbf{nonexecutable} \ \mathit{approach}(D) \ \textbf{if} \ \mathit{facing}(D). \end{array}$$

- $\mathit{gothrough}(D)$: the robot goes through door D. The robot can only go through a door if the door is accessible from the robot's current location, if it is open, and if the robot is facing it. Executing the *gothrough* action results in the robot's location being changed to the connecting room and the robot no longer faces the door.
- $\mathit{greet}(P)$: the robot greets person P. A robot can only greet a person if the robot knows that both the robot and that person are in the same room. Greeting a person P results in the $\mathit{visiting}(P)$ fluent being true.
- $\mathit{collectmail}(P)$: the robot collects mail from person P. A robot can only collect mail from a person if the robot knows that both the robot and that person are in the same room, if the person has not passed their mail to someone else, and if the person's mail has not been collected yet. Collecting mail from a person P results in the $\mathit{mailcollected}(P)$ fluent being true, formalized as

$$\mathit{collectmail}(P) \ \textbf{causes} \ \mathit{mailcollected}(P)$$

  Because of the recursive definition of *mailcollected* in (1), $\mathit{collectmail}(P)$ will also *indirectly* lead to the other people's mail passed to P to be collected as well.
- $\mathit{opendoor}(D)$: the robot opens a closed door D. The robot can only open a door that it is facing.

[3] An inertial fluent is a fluent whose value does not change with time by default.

- *askploc*(P): The robot asks the location of person P if it does not know the location of person P. Furthermore, the robot can only execute this action if it is visiting a person P_1 who knows the location of person P. This is the action that triggers human-robot interaction. By executing this action, the robot knows that the location of person P is room R, formalized as

$$\textit{askploc}(P_1, P)\ \textbf{causes}\ \textit{knowinside}(P, R)\ \textbf{if}\ \textit{inside}(P, R).$$

The above formalization can be easily written in the syntax of CPLUS2ASP, which translates it into the input language of the answer set solver CLINGO. The complete description is available with our code-release[1].

Planning with Action Description

Generating and executing plans

An action description in $\mathcal{BC}$ formalizes the domain as a transition system. In order to specify the planning problem, a *planning query* needs to be specified.

Before a plan can be generated, the planner needs to obtain an initial state from two sources:

- The planner maintains tables for some portion of the domain knowledge, namely, *knowinside*, *knows*, and *passto*, that help the robot reason about acquiring missing information and figure out how mail has been forwarded recursively. At planning time, the contents of the table are translated into a part of query that describes the initial state. For instance, the table that contains fluent values for *knowinside* is:

knowinside	o_1	o_2	o_3	lab_1
alice	**t**	**f**	**f**	**f**
bob	**f**	**t**	**f**	**f**
carol	**f**	**f**	**t**	**f**
dan	**f**	**f**	**f**	**f**

(2)

  Using this table, the planner outputs the table contents as a set of atoms which are joined to the query:

$$\textit{knowinside}(\textit{alice}, o_1), \sim\!\textit{knowinside}(\textit{alice}, o_2), \ldots,$$
$$\sim\!\textit{knowinside}(\textit{bob}, o_1), \textit{knowinside}(\textit{bob}, o_2), \ldots,$$

  It is important to note that all values in the last row of the table are **f**, indicating that Dan's location is not known.

- The planner polls the sensors to obtain the values of some portion of the time-dependent knowledge, namely, *beside*, *facing*, *open* and *loc*, and translates them into a part of the query that describes the initial state. The sensors guarantee that the value for *loc* is always returned for exactly one location, and *beside* and *facing* are returned with at most one door. If the robot is facing a door, the value of *open* for that door is sensed and returned as well. For instance, in the initial state, if the robot is in lab_1 and not facing any door, the planner senses and appends the following to the description of the initial state:

$$\textit{loc} = \textit{lab}_1, \sim\!\textit{beside}(d_4), \sim\!\textit{facing}(d_4), \ldots$$

In addition to the initial state, the query includes also a goal, for instance, *visiting*$(\textit{alice})$.

The planner uses CPLUS2ASP to translate the action description and query into a logic program following the syntax of answer set solver CLINGO, and then calls it to generate the answer sets. To find the shortest plan, CLINGO is called repeatedly with an incremental value of maximum plan length, up to a user-defined constant *maximumLength*. Execution is stopped at the first length for which a plan exists. A plan is represented as a sequence of actions and their time stamps. In the case where the robot starts in lab_1 and its goal is *visiting*$(\textit{alice})$, the following 7-step plan can satisfy the goal:

$$0\!:\!\textit{approach}(d_5), 1\!:\!\textit{opendoor}(d_5), 2\!:\!\textit{gothrough}(d_5),$$
$$3\!:\!\textit{approach}(d_1), 4\!:\!\textit{opendoor}(d_1), 5\!:\!\textit{gothrough}(d_1),$$
$$6\!:\!\textit{greet}(\textit{alice})$$

The output of CLINGO also contains the values of the fluents at various times:

$$0\!:\!\textit{loc} = \textit{lab}_1, 0\!:\!\sim\!\textit{facing}(d_5), 1\!:\!\textit{loc} = \textit{lab}_1, 1\!:\!\textit{facing}(d_5), \ldots$$

These fluents are used to monitor execution. Execution monitoring is important because, when using a real robot, it is possible that the action being currently executed by the robot does not complete successfully. In that case the robot may return observations different from the expected effects of the action. For instance, assume that the robot is executing *approach*(d_5) at time 0. The robot attempts to navigate to door d_5, but fails and returns an observation $\sim$*facing*(d_5) at time 1. Since this observation does not match the expected effect of *approach*(d_5) at time 1, which is *facing*(d_5), the robot incorporates $\sim$*facing*(d_5) as part of a new initial condition and plans again.

The Mail Collection Task

To solve the mail collection problem, the robot first needs to receive information about how mail was transferred from one person to another person, i.e. information that relates to the fluent *passto*. Any person who passes their mail to other people will send this information to the robot.

In our example domain, let's assume the robot receives the following information:

passto	*alice*	*bob*	*carol*	*dan*
alice	**f**	**f**	**f**	**f**
bob	**t**	**f**	**f**	**f**
carol	**f**	**f**	**f**	**f**
dan	**f**	**t**	**f**	**f**

(3)

Initially, let's assume that the robot is in lab_1 and not beside nor facing any door. The goal of collecting everyone's mail and reaching the corridor can be described as:

$$\textit{mailcollected}(alice), \textit{mailcollected}(bob),$$
$$\textit{mailcollected}(carol), \textit{mailcollected}(dan), loc = cor.$$

CLINGO generates an answer set with the following plan:

$$0\!:\!\textit{approach}(d_5), 1\!:\!\textit{opendoor}(d_5), 2\!:\!\textit{gothrough}(d_5),$$
$$3\!:\!\textit{approach}(d_1), 4\!:\!\textit{opendoor}(d_1), 5\!:\!\textit{gothrough}(d_1),$$
$$6\!:\!\textit{collectmail}(\textit{alice}),$$
$$7\!:\!\textit{approach}(d_1), 8\!:\!\textit{opendoor}(d_1), 9\!:\!\textit{gothrough}(d_1),$$
$$10\!:\!\textit{approach}(d_3), 11\!:\!\textit{opendoor}(d_3), 11\!:\!\textit{gothrough}(d_3),$$
$$13\!:\!\textit{collectmail}(\textit{carol}),$$
$$14\!:\!\textit{approach}(d_3), 15\!:\!\textit{opendoor}(d_3), 16\!:\!\textit{gothrough}(d_3)$$

In this plan, the robot only visits Alice and Carol, and doing so is sufficient to collect everyone's mail, even if Dan's location is not known.

Planning with Human Robot Interaction

Consider the modification of Table (3) in which Dan doesn't forward his mail to Bob. To collect Dan's mail, the robot now needs to visit him. However, the robot does not know where Dan is, as shown in the last row of Table (2). In our example domain, we assume Carol knows Dan's location:

knows	*alice*	*bob*	*carol*	*dan*
alice	**f**	**f**	**f**	**f**
bob	**f**	**f**	**f**	**f**
carol	**f**	**f**	**f**	**t**
dan	**f**	**f**	**f**	**f**

Again, let's assume that the robot is initially located in lab_1 and not beside nor facing any door. The planner calls CLINGO with the same initial state and same goal as in the previous section to generate the following shortest plan:

$0\!:\!approach(d_5), 1\!:\!opendoor(d_5), 2\!:\!gothrough(d_5),$
$3\!:\!approach(d_1), 4\!:\!opendoor(d_1), 5\!:\!gothrough(d_1),$
$6\!:\!collectmail(alice),$
$7\!:\!approach(d_1), 8\!:\!opendoor(d_1), 9\!:\!gothrough(d_1),$
$10\!:\!approach(d_3), 11\!:\!opendoor(d_3), 11\!:\!gothrough(d_3),$
$13\!:\!collectmail(carol),$
$14\!:\!greet(carol), 15\!:\!askploc(dan), 16\!:\!collectmail(dan),$
$17\!:\!approach(d_3), 18\!:\!opendoor(d_3), 19\!:\!gothrough(d_3)$

The first 13 steps of this plan are same as that of the plan generated in the previous section. It is important to notice that the answer set also contains the following fluent:

$$16\!:\!knowinside(dan, o_3) \tag{4}$$

This atom is the effect of executing action $askploc(dan)$ at time 15. Since CLINGO searches for the shortest plan by incrementing the number of steps, the "optimistic" plan that it finds corresponds to the case where Dan is located at the same office as Carol.

As before, the plan is executed and the execution is monitored. The robot executes action $askploc(dan)$ at time 15 by asking Carol for Dan's location. The robot obtains Carol's answer as an atom, for instance,

$$16\!:\!knowinside(dan, o_2),$$

which contradicts (4). As in the case of execution failure, replanning is necessary. Before replanning, the acquired information is used to update table (2). While replanning, the update table will generate a new initial condition that contains the following information $knowinside(dan, o_2)$.

After running CLINGO again, a new plan is found based on the information acquired from Carol (when replanning is triggered, the time stamp is reset to start from 0):

$0\!:\!approach(d_3), 1\!:\!opendoor(d_3), 2\!:\!gothrough(d_3),$
$3\!:\!approach(d_2), 4\!:\!opendoor(d_2), 5\!:\!gothrough(d_2),$
$6\!:\!collectmail(dan),$
$7\!:\!approach(d_2), 8\!:\!opendoor(d_2), 9\!:\!gothrough(d_2)$

By interacting with Carol, the robot obtained Dan's location, updated its knowledge base, and completed its goal. It should be noted that while planning under incomplete information can be achieved through sophisticated method such as conformant planning (Tu et al. 2011) and conditional planning (Son, Tu, and Baral 2004), our optimistic approach is extremely effective in acquiring missing information simply through execution monitoring and replanning.

Planning with Action Costs

Optimal Plan Generation

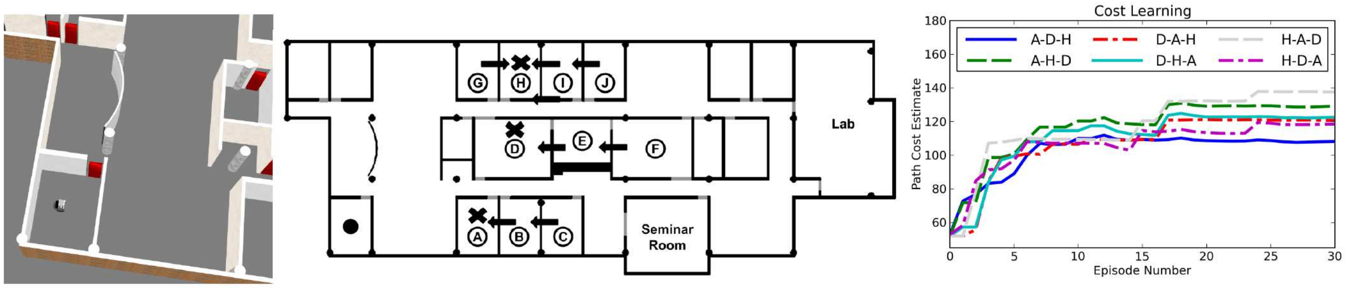
In the previous section, the planner generates multiple plans of equal length out of which one is arbitrarily selected for execution. In practice, those plans are not equivalent because different actions in the real world have different costs. In our domain, we consider the cost of an action to be the time spent during its execution. For instance, when the robot visits Alice in the first few steps of plan execution, the generated plan includes the robot exiting lab_1 through door d_5. The planner also generated another plan of the same length where the robot could have exited through door d_4, but that plan was not selected. If we see the layout of the example environment in Figure 2, we can see that it is indeed faster to reach Alice's office o_1 through door d_4. In this section, we present how costs can be associated with actions such that a plan with the smallest cost can be selected to achieve the goal.

Costs are functions of both the action being performed and the state at the beginning of that action. CPLUS2ASP does not directly support formalizing costs for generating optimal plans, but CLINGO allows the user to write a logic program with *optimization statements* (indicated via the keywords `#maximize` and `#minimize`) to generate optimal answer sets. Therefore, in our application, cost formalization and optimization statements are directly written in logic program rules in CLINGO syntax. They are then appended to the domain description and the query, and sent to CLINGO to generate an optimal plan. In the example domain, for simplicity, we assume all actions to have fixed costs apart from *approach*. Actions *askploc*, *opendoor*, *greet*, and *collectmail* have cost 1, and *gothrough* has cost 5.

The costs for executing action $approach(D)$ depend on the physical location of the robot. It is computed in two different ways:

- When the robot approaches door D_1 from door D_2 and is currently located in R, the values of fluents uniquely identify the physical location of the robot in the environment. The cost of action $approach(D_1)$ is specified by an *external term* `@cost(D1,D2,R)` supported by CLINGO. At the time of plan generation, CLINGO will make external function calls to compute the value of external term.
- When the robot is not next to a door, for instance, in the middle of the corridor, the cost for approaching any door is fixed to 10. This only happens in the initial condition.

With all actions associated with costs we use the optimization statement `#minimize` to guide the solver to return a plan of optimal cost, instead of shortest length. Different from the previous case without costs, we do not call CLINGO repeatedly with incremental values of maximum plan length, and directly search for the optimal plan with a maximum of *maximumLength* steps. It is important to note that the optimal plan found by CLINGO is not necessarily the global optimal plan, but only the optimal plan up to *maximumLength* steps. *maximumLength* needs to be set appropriately to balance optimality with execution time based on computational power available.



(a) Simulation in GAZEBO **(b)** Floor plan **(c)** Learning Results

Figure 3: The simulation domain contains 20 rooms, 25 doors and 10 people from whom mail has to be collected. The filled circle marks the robot's start position, and the crosses mark the people who hold all the mail (A, D, H), and the arrows mark how mail was recursively passed to them. Figure 3c shows the costs of 6 plans as actions costs are learned in the environment.

Estimating costs through environment interactions

Although costs for different navigation actions can be estimated from the dedicated low-level navigation module, the navigation module can only return costs based on spatial metrics. In this work, we use navigation time instead of distance as the cost measure, and estimate this time through interactions with the environment. Whenever the executor successfully performs an action, the cost estimator gets a *sample* of the true cost for that action. It then updates the current estimate for that action using an *exponentially weighted moving average*:

$$cost_{e+1}(X, Y) = (1 - \alpha) \times cost_e(X, Y) + \alpha \times sample$$

where e is the episode number, α is the learning rate and set to 0.5 in this paper, X is the action, and Y is the initial state.

To apply this learning rule, we need estimates of all costs at episode 0. Since we want to explore a number of plans before choosing the lowest-cost one, we use the technique of *optimistic initialization* (Sutton and Barto 1998) and set all initial cost estimates to a value which is much less than the true cost. This causes the robot to underestimate the cost of an action it has not taken often enough for its estimate to converge to the true value. The exploration in optimistic initialization is short-lived (Sutton and Barto 1998). Once the cost estimator sets the values such that a particular plan becomes larger than the current best plan, the planner will never attempt to follow that plan even though its costs may decrease in the future. There are known techniques such as ϵ-greedy exploration in the literature (Sutton and Barto 1998) that attempt to solve this problem. We leave testing and evaluation of these approaches to future work.

Experiments

We evaluate planning and learning in a domain using both a real robot and a realistic simulation of that robot. The robot used in these experiments is built on top of a Segway RMP, and uses a Hokuyo URG-04LX LIDAR and a Kinect RGB-D camera for navigation and sensing. Autonomous navigation on the robot is based on the *elastic band* approach (Quinlan and Khatib 1993).

Actions requested by the planner are mapped into specific low-level procedures on the robot as follows:

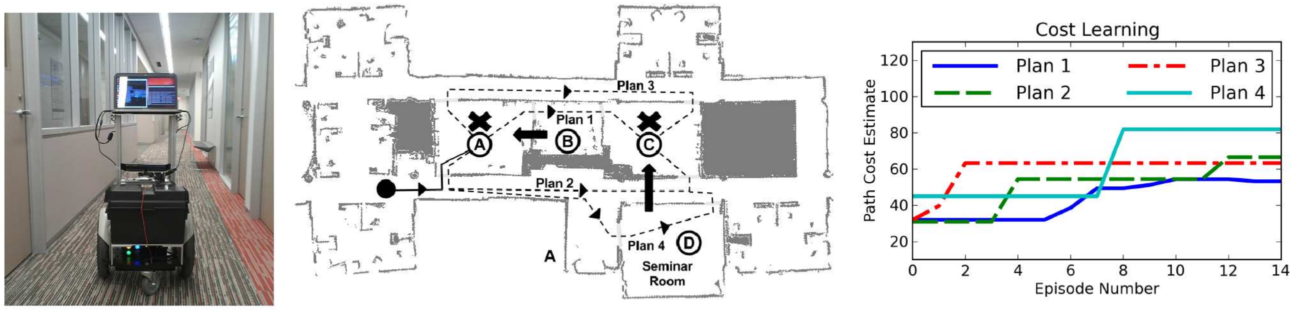
- The *approach* action autonomously navigates the robot to a prespecified location approximately $1m$ from the door.
- The *gothrough* action navigates the robot to a similarly prespecified location on the other side of the door.
- The *opendoor* action automatically opens the door in simulation. In the real world, the robot does not have the capability to open the door itself, and requests a human to open the door instead.
- The *askploc* action requests the location of a person, and the answer is input by the human from the terminal. The answer is then encoded using the *knowinside* fluent and returned to the planner.
- The *greet* and *collectmail* actions are simply placeholders, and pause the robot for a short duration during which the robot uses a speech synthesis module to greet the person and request mail.

The observations from the robot's sensors are grounded into valued fluents as follows:

- The *beside* fluent is true if the robot is within $2m$ of the door. If the robot is sufficiently close to multiple doors, it will only set *beside* for a single door.
- The *facing* fluent is true if the robot is beside the door, and the orientation of the robot does not differ from the orientation to the door by more than $\pi/3$.
- The *open* door fluent is true if the robot senses that it can navigate through the door.
- The *loc* fluent is mapped from the physical location of the robot to a logical location using a look-up table. The real robot estimates its position using *adaptive Monte Carlo localization* (Fox et al. 1999).

Simulation Experiments

Experiments in simulation test our approach in the domain illustrated in Figure 3. This domain has a total of 10 people from whom mail needs to be collected, 20 rooms and 25 doors. Action execution inside the simulator is typically successful. In order to demonstrate how learning can adapt planning and allow for the generation of optimal plans, we learn navigation costs through multiple episodes on a single problem instance where all the mail has been passed to 3 people (A, D, H in Figure 3b), and the robot starts at location indicated by the filled circle.



(a) Segway based Robot **(b)** Floor plan **(c)** Learning Results

Figure 4: The real world domain contains 5 rooms, 8 doors and 4 people from whom mail has to be collected. The filled circle marks the robot's start position, and the crosses mark the people who have all the mail (A, C), and the arrows mark how mail was recursively passed to them. The 4 plans compared in Figure 4c are also marked on the floor plan.

All simulation experiments were run on a machine with a Quad-Core i7-3770 processor, where the processing was split between 3D simulation, visualization and the planner. CLINGO ran using 6 parallel threads to generate plans, and searched for the optimal plan among all plans up to a length of 35 steps (*maximumLength*). After 1 minute of planning, the best available plan from CLINGO was selected for execution. Since this domain contains a large number of correct but sub-optimal plans, we restrict the size of the search space to decrease the number of episodes required to learn the costs for the optimal plan. This reduction is achieved by appending the following heuristics to the query:

- Approach a door only if the next action goes through it.
- Don't go through a door if the next action is to go back through it again without taking any other action.

Since there are a large number of correct plans, we only present the cost curves for six plans to demonstrate learning. These six plans correspond to the best plans for all permutations of the order in which mail is collected from A, D, and H. The plan that collects mail in the order A-D-H is optimal. Figure 3c shows the total costs of these 6 plans as the cost estimates improve. By episode 17, the costs are learned sufficiently well such that the planner converges to the true optimum plan A-D-H. However, it should be noted that since the planner may be terminated early before finding this plan, on occasion other plans may be selected for execution. For instance, in episode 24, the plan H-D-A gets executed, as shown by the significant increase in the cost values of the 2 plans H-D-A and H-A-D in Figure 3c. This problem can be alleviated by allowing longer planning times.

Real World Experiments

Experiments using the real robot have been run in the same environment on which the simulation in the previous section was based on. Since tests on the real robot require considerably more effort, and robot execution cannot be sped up, the real world domain has been reduced to a subset of the simulation domain. The real world domain contains 5 rooms, 8 doors, and 4 people from whom mail has to be collected. 2 people have passed mail forwards such that the robot only needs to visit a total of 2 people. The domain is illustrated in Figure 4. Since the domain is smaller than that in simulation, the planner can typically generate the optimal plan in 10-15 seconds and verify that it is optimal. It should be noted that execution in the real world often results in failure, and replanning occurs frequently.

We present the cost curves of 4 different plans in Figure 4c, where Plan 1 is optimal. In this experiment, the robot starts in the middle of the corridor not beside any door as shown in Figure 4b. Consequently, the cost of the first navigation action cannot be learned as the true physical location of the robot gets abstracted away. The learning curves shows that the planner discovers by the episode 12 that plan 1 is optimal. Different from the simulation experiment there is no early termination of the planner. After the optimal plan is found, no other plans are selected for execution and their costs don't change. In addition to the quantitative evaluation in this section, we also present a qualitative evaluation of the mail collection task in a online video appendix[1].

Conclusion

In this paper, we introduced an approach that uses action language $\mathcal{BC}$ for robot task planning, and incorporates action costs to produce optimal plans. We applied this approach to a mail collection task using a real robot, as well as a realistic 3D simulator. Using action language $\mathcal{BC}$ allows us to formalize indirect effects of actions on recursive fluents. In the presence of incomplete information, the proposed approach can generate plans to acquire missing information through human-robot interaction. Furthermore, by estimating costs from experience, we can adapt planning while learning costs in the environment.

Acknowledgments

The authors would like to thank ROS, GAZEBO, and CLINGO developers for infrastructure used in this work. The authors would also like to thank Chien-Liang Fok, Sriram Vishwanath and Christine Julien for their assistance in constructing the Segway robot.

A portion of this research has taken place in the Learning Agents Research Group (LARG) at the AI Laboratory, UT Austin. LARG research is supported in part by grants from the NSF (CNS-1330072, CNS-1305287), ONR (21C184-01), and Yujin Robot. LARG research is also supported in part through the Freshman Research Initiative (FRI), College of Natural Sciences, UT Austin.

References

Babb, J., and Lee, J. 2013. Cplus2ASP: Computing action language $\mathcal{C}+$ in answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*.

Balduccini, M.; Gelfond, M.; and Nogueira, M. 2006. Answer set based design of knowledge systems. In *Annals of Mathematics and Artificial Intelligence*.

Blockeel, H., and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artificial intelligence (AIJ)*.

Caldiran, O.; Haspalamutgil, K.; Ok, A.; Palaz, C.; Erdem, E.; and Patoglu, V. 2009. Bridging the gap between high-level reasoning and low-level control. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*.

Chen, X.; Ji, J.; Jiang, J.; Jin, G.; Wang, F.; and Xie, J. 2010. Developing high-level cognitive functions for service robots. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Chen, X.; Jin, G.; and Yang, F. 2012. Extending $\mathcal{C}+$ with composite actions for robotic task planning. In *International Conference on Logical Programming (ICLP)*.

de la Rosa, T.; Olaya, A. G.; and Borrajo, D. 2007. Using cases utility for heuristic planning improvement. In *International Conference on Case-based Reasoning (ICCBR)*.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003. Answer set planning under action costs. *Journal of Artificial Intelligence Research (JAIR)*.

Erdem, E.; Aker, E.; and Patoglu, V. 2012. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics (ISR)*.

Erdem, E., and Patoglu, V. 2012. Applications of action languages in cognitive robotics. In Erdem, E.; Lee, J.; Lierler, Y.; and Pearce, D., eds., *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*. Springer.

Erdem, E.; Patoglu, V.; Saribatur, Z. G.; Schüller, P.; and Uras, T. 2013. Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. *Theory and Practice of Logic Programming (TPLP)*.

Finger, J. 1986. *Exploiting Constraints in Design Synthesis*. Ph.D. Dissertation, Stanford University.

Fox, D.; Burgard, W.; Dellaert, F.; and Thrun, S. 1999. Monte carlo localization: Efficient position estimation for mobile robots. In *National Conference on Artificial Intelligence (AAAI)*.

Gebser, M.; Kaminski, R.; König, A.; and Schaub, T. 2011. Advances in *gringo* series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*.

Gebser, M.; Grote, T.; and Schaub, T. 2010. Coala: a compiler from action languages to ASP. In *European Conference on Logics in Artificial Intelligence (JELIA)*.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *International Logic Programming Conference and Symposium (ICLP/SLP)*.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing*.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence (ETAI)*.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence (AIJ)*.

Havur, G.; Haspalamutgil, K.; Palaz, C.; Erdem, E.; and Patoglu, V. 2013. A case study on the Tower of Hanoi challenge: Representation, reasoning and execution. In *International Conference on Robotics and Automation (ICRA)*.

Jiménez, S.; Fernández, F.; and Borrajo, D. 2013. Integrating planning, execution, and learning to improve plan execution. *Computational Intelligence*.

Koenig, N., and Howard, A. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *International Conference on Intelligent Robots and Systems (IROS)*.

Lee, J.; Lifschitz, V.; and Yang, F. 2013. Action language $\mathcal{BC}$: A preliminary report. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Leonetti, M.; Iocchi, L.; and Patrizi, F. 2012. Automatic generation and learning of finite-state controllers. In *Artificial Intelligence: Methodology, Systems, and Applications*. Springer. 135–144.

Marek, V., and Truszczynski, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag.

McCain, N., and Turner, H. 1997. Causal theories of action and change. In *National Conference on Artificial Intelligence (AAAI)*.

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*. Edinburgh University Press.

McCarthy, J. 1987. Generality in Artificial Intelligence. *Communications of the ACM (CACM)*.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. ROS: an open-source robot operating system. In *Open Source Softare in Robotics Workshop at ICRA '09*.

Quinlan, S., and Khatib, O. 1993. Elastic bands: Connecting path planning and control. In *International Conference on Robotics and Automation (ICRA)*.

Quintero, E.; Alcázar, V.; Borrajo, D.; Fernández-Olivares, J.; Fernández, F.; Garcia-Olaya, A.; Guzmán, C.; Onaindia, E.; and Prior, D. 2011a. Autonomous mobile robot control and learning with the PELEA architecture. In *Automated Action Planning for Autonomous Mobile Robots Workshop at AAAI '11*.

Quintero, E.; Garcia-Olaya, Á.; Borrajo, D.; and Fernández, F. 2011b. Control of autonomous mobile robots with automated planning. *Journal of Physical Agents (JoPhA)*.

Reiter, R. 1978. On closed world data bases. In *Logic and Data Bases*. Plenum Press.

Son, T. C.; Tu, P. H.; and Baral, C. 2004. Planning with sensing actions and incomplete information using logic programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*. Cambridge University Press.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of PDDL axioms. In *International Joint Conferences on Artificial Intelligence (IJCAI)*.

Tu, P. H.; Son, T. C.; Gelfond, M.; and Morales, A. R. 2011. Approximation of action theories and its application to conformant planning. *Artif. Intell.* 175(1):79–119.

Yang, F.; Khandelwal, P.; Leonetti, M.; and Stone, P. 2014. Planning in answer set programming while learning action costs for mobile robots. In *AAAI Spring 2014 Symposium on Knowledge Representation and Reasoning in Robotics (AAAI-SSS)*.