To appear in AAAI Spring 2014 Sympoisum on Knowledge Representation and Reasoning in Robotics (AAAI-SSS), March 2014.

Planning in Answer Set Programming while Learning Action Costs for Mobile Robots

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

Abstract

For mobile robots to perform complex missions, it may be necessary for them to plan with incomplete information and reason about the indirect effects of their actions. Answer Set Programming (ASP) provides an elegant way of formalizing domains which involve indirect effects of an action and recursively defined fluents. In this paper, we present an approach that uses ASP for robotic task planning, and demonstrate how ASP can be used to generate plans that acquire missing information necessary to achieve the goal. Action costs are also incorporated with planning to produce optimal plans, and we show how these costs can be estimated from experience making planning adaptive. We evaluate our approach using a realistic simulation of an indoor environment where a robot learns to complete its objective in the shortest time.

Introduction

Automated planning provides great flexibility over direct implementation of behaviors for robotic tasks. In mobile robotics, uncertainty about the environment stems from many sources. This is particularly true for domains inhabited by humans, where the state of the environment can change outside the robot's control in ways hard to predict. Probabilistic planners attempt to capture this complexity, but planning in probabilistic representations makes reasoning much more computationally expensive. Furthermore, correctly modeling the system using probabilities is more difficult than representing knowledge symbolically. Indeed, robotic planning systems are frequently based on symbolic deterministic models, and execution monitoring. The brittleness owing to errors in the model and unexpected conditions are overcome through monitoring and replanning.

Automated symbolic planning includes early work such as situational calculus (McCarthy and Hayes 1969), STRIPS (Fikes and Nilsson 1971), ADL (Pednault 1989); recent action languages such as $\mathcal{C}+$ (Giunchiglia et al. 2004) and $\mathcal{BC}$ (Lee, Lifschitz, and Yang 2013); and declarative programming languages such as Prolog and logic programming based on answer set semantics (Gelfond and Lifschitz 1988; 1991). The latter is also referred to as Answer Set Programming (ASP). These languages allow users to formalize dynamic domains as transition systems, by a set of axioms that specify the precondition and effects of actions and the relationship between state variables. In ASP, the frame problem (McCarthy and Hayes 1969) can be solved by formalizing the commonsense law of inertia.

Compared to STRIPS or PDDL-style planning languages, ASP and action languages provide an elegant way of formalizing indirect effects of actions as *static laws*, and thus solve the ramification problem (Lin 1995). Although the original specification of PDDL includes axioms (which correspond to non-recursive static laws in our terminology), the semantics for these axioms are not clearly specified. Indeed, in most planning domains investigated by the planning community and in planning competitions, no axioms are needed. *Thiébaux et al.* (Thiébaux, Hoffmann, and Nebel 2003) argued that the use of axioms not only increases the expressiveness and elegance of the problem representation, but also improves the performance of the planner. Indeed, in recent years, robotic task planning based on ASP or related action languages has received increasing attention (Caldiran et al. 2009; Erdem and Patoglu 2012; Chen et al. 2010; Chen, Jin, and Yang 2012; Erdem et al. 2013; Havur et al. 2013).

Incorporating costs in symbolic planning (Eiter et al. 2003) is important for applications that involve physical systems that have limited resources such as time, battery, communication bandwidth etc. In this paper, we show how costs can be incorporated into a robot planning system based on ASP, and how these costs can be learned from experience. It is important to learn costs from the environment, since these costs may not be 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 domain formalization. As such, the main contribution of this paper is an approach that uses ASP for robot task planning while learning costs of individual actions through experience.

We evaluate our approach using a simulated mobile robot navigating through an indoor environment and interacting with people. Should certain information necessary for completing a task be missing, the planner can issue a sensing action to acquire this information through human-robot interaction. By integrating the planner with a learning module, the costs of actions are learned from plans' execution. Since the only interface between the planner and the learning module is through costs, the action theory is left unchanged, and the proposed approach can in principle be applied to any metric planner. All the code used in this paper has been implemented using the ROS middleware package (Quigley et al. 2009), and the realistic 3D simulator GAZEBO (Koenig and Howard 2004), and is available in the public domain[1].

Related Work

In recent years, action languages and answer set programming have begun to be used for robot task planning (Caldiran et al. 2009; Erdem and Patoglu 2012; Erdem et al. 2013; Havur et al. 2013). In these previous works, domains are small and idealistic. The planning paradigm is typical classical planning with complete information, and shortest plans are generated. Since the domains are small, it is possible to use grid worlds so that motion control is also achieved by planning. In contrast, we are interested in large and continuous domains, where navigation is handled by a dedicated module to achieve continuous motion control, and require planning with incomplete information by performing knowledge acquisition through human-robot interaction.

The work of *Erdem et al.* (Erdem, Aker, and Patoglu 2012) 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. As such, their work attempts to find the shortest plan that satisfies the goal within this time constraint. In contrast, our work attempts to explicitly minimize the overall cost to execute the plan (i.e. the optimal plan), instead of finding the shortest plan. Another difference between the two approaches is that *Erdem et al.* attempt to include geometric reasoning at the planning level, i.e. the ASP solver considers a coarsely discretized version of the true physical location of the robot. Since we target larger domains, we discretize the location of the robot at the room level 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. The approach most closely related to ours is the PELA architecture (Jimnez, Fernndez, 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. We estimate costs based on any metric observable by the robot. Some preliminary work has been done in PELA to learn expected action durations (Quintero et al. 2011), using a variant of relational decision trees. Differently, we learn costs simply using exponentially weighted averaging, which allows us to respond to recent changes in the environment. Furthermore, while that work is specific to PDDL-style planners, we perform the first integration between ASP and learning, at the advantage of defining indirect effects of actions and recursively defined fluents as planning heuristics.

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

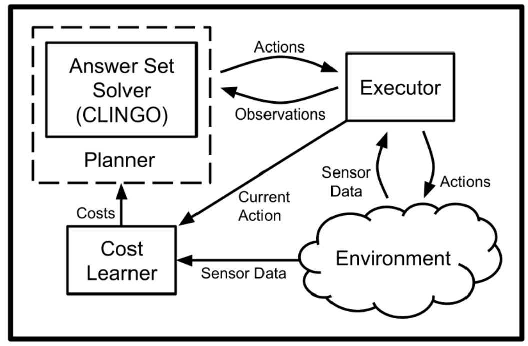


Figure 1: The architecture implemented in our approach. The planner invokes a cost estimator that learns the actual action costs from the sensor data during execution.

Architecture Description

Our proposed architecture (shown in Figure 1) has two modules that comprise the decision making: a planning module, and a cost estimation module. At planning time, the planner polls the cost of each action of interest, and produces a minimum plan. The planner itself is constituted by a domain description specified in ASP and a solver, in our case CLINGO (Gebser et al. 2011a). After plan generation, an executor invokes the appropriate controllers for each action, and grounds numeric sensor data into symbolic fluents, for the planning module to verify. If the observed fluents are incompatible with the state expected by the planner, replanning is triggered. During action execution, the cost estimator receives the sensor data, and employs a learning algorithm to estimate the value of the expected cost of each action from the experienced samples.

This architecture allows to treat the planning module as a black box, and can in principle be adopted with any metric planner. For the same reason, the formalism used to represent the state space in the planner and the cost estimator modules need not be same. In the following sections, we give a detailed description of the different elements of the system.

Domain Representation

In order to demonstrate how ASP can be used for robot task planning under incomplete information and with action costs, we consider a domain where a mobile robot navigates inside a building, visiting and serving the inhabitants. In this section, we use a small floor plan, illustrated in Figure 2, to clearly explain the domain formalization process. In the experimental section, we will evaluate our approach on a domain with a much larger floor plan based on a real building.

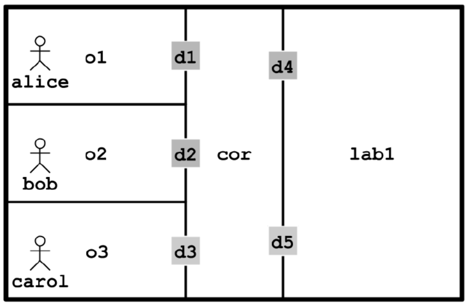


Figure 2: The layout of the example floor plan with an overlay of the rigid knowledge provided to the planner.

Formalizing the Dynamic Domain

Domain knowledge about a building includes the following four different types of information: *rigid knowledge*, *time-dependent external knowledge*, *time-dependent internal knowledge* and *action knowledge*. We explain each of these in detail in the following subsections. All rules formalizing aspects of the domain are presented using syntax from GRINGO (Gebser et al. 2011b). Due to space constraints, some portions of the ASP encoding are not present and can be viewed online in our code base.

Rigid knowledge includes information about the building that does not depend upon the passage of time. In other words, as the robot moves around inside the building, rigid knowledge should not change. In our example, rigid knowledge includes information about offices, labs, corridors, doors, and accessibility between all these locations. It also includes information about building residents, their occupations, the offices to which they are assigned, and whether a person knows where another person is. Rigid knowledge has been formalized in our system using the following predicates:

- `office(X)`, `door(Y)`, `room(Z)`: `X` is an office, `Y` is a door and `Z` is a room.
- `hasdoor(X,Y)`: room `X` has door `Y`.
- `acc(X,Y,Z)`: room `X` is accessible from room `Z` via door `Y`.
- `indacc(X,Y,Z)`: room `X` is indirectly accessible from room `Z` via door `Y`.
- `secretary(X)`, `faculty(Y)`, `person(Z)`: `X` is a secretary, `Y` is a faculty, and `Z` is a person.
- `in_office(X,Y)`: person `X` is assigned to office `Y`.
- `knows(X,ploc(Y))`: person `X` knows the location of person `Y`.

The following rules define the rigid knowledge in the example environment illustrated in Fig. 2:

```
room(cor).
office(o1;;o2;;o3;;lab1).
room(X):- office(X).
door(d1;;d2;;d3;;d4;;d5).
hasdoor(o1,d1;;o2,d2;;o3,d3;;
        lab1,d4;;lab1,d5).
secretary(carol).
faculty(alice;;bob).
in_office(alice,o1;;bob,o2;;carol,o3).
knows(carol,ploc(X)):- faculty(X).
person(X) :- faculty(X).
person(X) :- secretary(X).
```

We also define additional rules that define accessibility information between 2 rooms:

```
acc(X,Y,cor):- room(X),door(Y),hasdoor(X,Y).
  acc(Z,Y,X):- acc(X,Y,Z).
indacc(X,Y,Z):- acc(X,Y,Z).
indacc(X,Y,Z):- acc(X,Y,Z1),indacc(Z1,Y1,Z).
```

It is important to notice that `indacc` is recursively defined by the last rule above, which is usually not easily achieved in a PDDL-style language. Later we will show that `indacc` is used to formulate *planning heuristics* which significantly shorten planning time.

Time-dependent external knowledge includes information about the environment that can change over time and cannot be affected by the robot's actions. For the example environment, time-dependent external knowledge is formalized as follows:

- The current location of a person is formalized by the fluent `inside`. `inside(X,Y,I)` means at time instant `I`, person `X` is located in room `Y`.
- By default, a person is inside the office that he is assigned to:

  ```
  inside(X,Y,0):- in_office(X,Y),
                  not -inside(X,Y,0).
  ```

  In the above rule we used two kinds of negation: `-` is known as strong negation (classical negation) in ASP literature (Gelfond and Lifschitz 1991), which is the same as the negation sign ¬ in classical logic, and `not` is known as *negation as failure* (Gelfond and Lifschitz 1988), which is similar to the negation in Prolog and intuitively understood as *there is no evidence.*
- A person can only be inside a single room at any given time:

  ```
  -inside(X,Y,I):- inside(X,Y1,I), Y1!=Y,
                   room(Y).
  ```
- `inside` is an *inertial fluent*. An inertial fluent is a fluent whose value does not change by default. These two rules formalize the commonsense law of inertia for the fluent `inside`:

  ```
  inside(X,Y,I+1):- inside(X,Y,I),
                    not -inside(X,Y,I+1), I<n.
  -inside(X,Y,I+1):- -inside(X,Y,I),
                    not  inside(X,Y,I+1), I<n.
  ```

  where `n` denotes maximum possible time steps. To solve a planning task, the value of n specifies the upper-bound on the number of steps in the plan.

Time-dependent internal knowledge describes the fluents that are directly affected by the robot's actions. In the example domain, time-dependent internal knowledge is formalized through the following predicates and rules:

- `open(X,I)`: a door `X` is open at time `I`. By default, a door is not open.

```
-open(X,I):- not open(X,I), door(X).
```

- `facing(X,I)`: the robot is next to and facing door `X` at time `I`. The fluent is inertial. The robot cannot face two different doors simultaneously.

```
 facing(X,I+1):- facing(X,I),
                 not -facing(X,I+1), I<n.
-facing(X,I+1):- -facing(X,I),
                 not  facing(X,I+1), I<n.
 -facing(X2,I):- facing(X1,I), X1!=X2,
                    door(X2),I<=n.
```

- `beside(X,I)`: the robot is next to door `X` at time `I`. `beside` is implied by `facing`, but not the other way around. The fluent is inertial. The robot cannot be beside two different doors simultaneously.

```
   beside(X,I):- facing(X,I).
 beside(X,I+1):- beside(X,I),
                 not -beside(X,I+1), I<n.
-beside(X,I+1):- -beside(X,I),
                 not  beside(X,I+1), I<n.
 -beside(X2,I):- beside(X1,I), X1!=X2,
                 door(X2),I<=n.
```

Since `beside` is implied by `facing`, it is an indirect effect of any actions that affect `facing`.

- `at(X,I)`: the robot is at room `X` at time `I`. `at` is inertial, and the robot must be at exactly one location at a given time.

```
{at(X,I+1)}:- at(X,I), room(X), I<n.
           :- not 1{at(X,I):room(X)}1.
```

- `visiting(X,I)`: the robot is visiting a person `X` at time `I`. By default, a robot is not visiting anyone.

```
-visiting(X,I):- not visiting(X,I),
                 person(X).
```

Action knowledge includes the rules that formalize robot actions, the preconditions to execute these actions, and the effects of these actions. Actions are divided into *non-sensing* and *sensing* actions. Non-sensing actions change the state of the world, for instance a robot can approach a door and change its own location. On the other hand, sensing actions don't change the state of the world and are executed to acquire missing information. The robot has four non-sensing actions:

- `approach(X,I)`: the robot approaches door `X` at time `I`. A robot can only approach a door accessible from its current location if it is not facing the door already. Approaching a door should result in the robot facing that door.

```
:- approach(Y,I), at(X,I),
              {door(Y):acc(X,Y,Z)}0, I<n.
:- approach(X,I), facing(X,I), I<n.
facing(X,I+1):- approach(X,I),door(X),I<n.
```

- `gothrough(X,I)`: the robot goes through door `X` at time `I`. 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.

```
:- gothrough(Y,I), at(X,I),
               {door(Y):acc(X,Y,Z)}0, I<n.
:- gothrough(Y,I), not open(Y,I), I<n.
:- gothrough(Y,I), not facing(Y,I), I<n.
```

Executing the `gothrough` action results in the robot's location being changed to the connecting room and the robot no longer facing the door.

```
      at(Z,I+1):- gothrough(Y,I),
                  at(X,I), acc(X,Y,Z), I<n.
-facing(Y,I+1):- gothrough(Y,I),
                  at(X,I), acc(X,Y,Z), I<n.
```

- `greet(X,I)`: the robot greets person `X` at time `I`. A robot can only greet a person if both the robot and that person are in the same room. Greeting a person results in the `visiting` fluent being true.

```
:- greet(X,I), at(Y,I), inside(X,Y1,I),
                        Y!=Y1, I<n.
  visiting(X,I+1):- greet(X,I), I<n.
```

- `opendoor(X,I)`: the robot opens a closed door `X` at time `I`. The robot can only open a door that it is facing.

```
:- opendoor(X,I), not facing(X,I), I<n.
:- opendoor(X,I), open(X,I), I<n.
open(X,I+1):-opendoor(X,I),-open(X,I),I<n.
```

The robot has one sensing action:

- `askploc(X,I)`: The robot asks the location of person `X` at time `I` if it does not know the location of person `X`. Furthermore, the robot can only execute this action if it is visiting a person `Y` who knows the location of person `X`. The resulting state should include the location of person `X` entered by person `Y` as room `Z`.

```
:- askploc(X,I), inside(X,Y,I), I<n.
:- askploc(X,I),
          {visiting(Y,I):person(Y)}0, I<n.
:- askploc(X,I), visiting(Y,I),
                 not knows(Y,ploc(X)), I<n.
1{inside(X,Z,I+1):room(Z)}1:-
                        askploc(X,I), I<n.
```

Also included are a set of constraints that forbid concurrent execution of actions and a set of choice rules such that at any time, executing any action is arbitrary. We present only one example of a choice rule below which formalizes that at any time `I<n`, the robot has the right to approach a door `X`.

```
{approach(X,I)}:- door(X), I<n.
```

Generating and executing plans

The planner first queries the robot for its initial state, which is returned through sensing in the form of observable fluent values for `at`, `beside`, `facing` and `open`:

```
at(lab1,0) -beside(d4) -beside(d5) ...
```

The sensors guarantee that the values for `at` is always returned for exactly one location, and `beside` and `facing` are returned with at most one door.

With the initial state available, the planner uses the answer set solver CLINGO to plan the steps for achieving a formally described goal. For instance, if the goal is to visit Bob, then the goal is formalized as:

```
:- not visiting(bob,n).
```

To find the shortest plan for this goal, we iteratively increment `n` (starting at 1) and call the answer set solver to see if a plan exists. Since we are interested in a plan of reasonable size, we assign an upper bound to `n`, denoted by `max_size`. If a plan is not generated within this upper bound, we declare that the goal cannot be achieved. Assuming that the robot starts in `lab1` and does not face a door initially, the answer set solver successfully finds the following plan when `n=7`:

```
approach(d4,0) opendoor(d4,1)
gothrough(d4,2) approach(d2,3)
opendoor(d2,4) gothrough(d2,5)
greet(bob,6)
```

These atoms represent the plan and are executed on the robot one at a time. The answer set also contains atoms that represent world transitions, such as:

```
at(lab1,0) -open(d4,0) -facing(d4,0)
at(lab1,1) -open(d4,1) facing(d4,1)
at(lab1,2) open(d4,2) facing(d4,2)
...
```

These atoms can be used to monitor execution. Execution monitoring is important since 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, let's assume that the robot is currently executing `approach(d4,0)`. The robot attempts to navigate to door `d4`, but fails and returns an observation `-facing(d4,1)`. Since this observation does not match the expected effect of `approach(d4,t)`, which is `facing(d4,1)`, the robot incorporates `-facing(d4,0)` as part of a new initial state and replans.

Planning with incomplete information

Planning can also account for incomplete information, as the robot can sense missing information from the environment. Consider a visiting faculty named Dan who is inside the building. The robot needs to visit Dan, but does not know where he is. However, the robot is aware that Carol knows the location of all faculty inside the building, and the generated plan includes visiting her to acquire Dan's location. The goal of such a plan is described below:

```
faculty(dan).
:- not visiting(dan, n).
```

We call CLINGO with this goal to generate the shortest plan. Assuming again that the robot is inside `lab1`, when `n=9`, CLINGO returns the answer set which contains the following actions:

```
approach(d4,0) opendoor(d4,1)
gothrough(d4,2) approach(d3,3)
opendoor(d3,4) gothrough(d3,5)
greet(carol,6) askploc(dan,7)
greet(dan,8)
```

It is important to note that the robot greets Dan immediately after asking Carol for his location, without moving to a different room. This plan is returned because the answer set contains the following atom:

```
inside(dan,d3,8)
```

This atom is the effect of executing sensing action `askploc(dan,7)`. Since we search for the shortest plan by incrementing the value of steps `n`, Dan being located in the same office as Carol facilitates the generation of the shortest plan. Therefore, at plan generation time, this missing information is *assumed optimistically*.

As before, the plan is executed and the execution is monitored. The robot executes action `askploc(dan,7)` by asking Carol for Dan's location. The robot obtains Carol's answer as an atom, for instance, `inside(dan,o1,8)`, which contradicts the optimistic assumption. Similar to execution failures, a replan is called with a new initial state that contains:

```
inside(dan,o1,0)
```

After running CLINGO again, a new plan is found:

```
approach(d3,0) opendoor(d3,1)
gothrough(d3,2) approach(d1,3)
opendoor(d1,4) gothrough(d1,5)
greet(dan,6)
```

It is important to note that in this scenario, formalizing sensing action `askploc` is essential to achieve the task of visiting Dan. Without this action, no plan can be found because no other action can generate knowledge about Dan's location. On the other hand, even if `askploc` is formalized to have multiple effects, only those that contribute to finding the shortest plans can occur in the plan, and the plan is revised during execution. This paradigm is different from conditional planning with sensing actions (Son, Tu, and Baral 2004), where all outcomes of sensing actions are generated off-line to obtain a tree-shaped plan, which is exponentially larger than a linear plan.

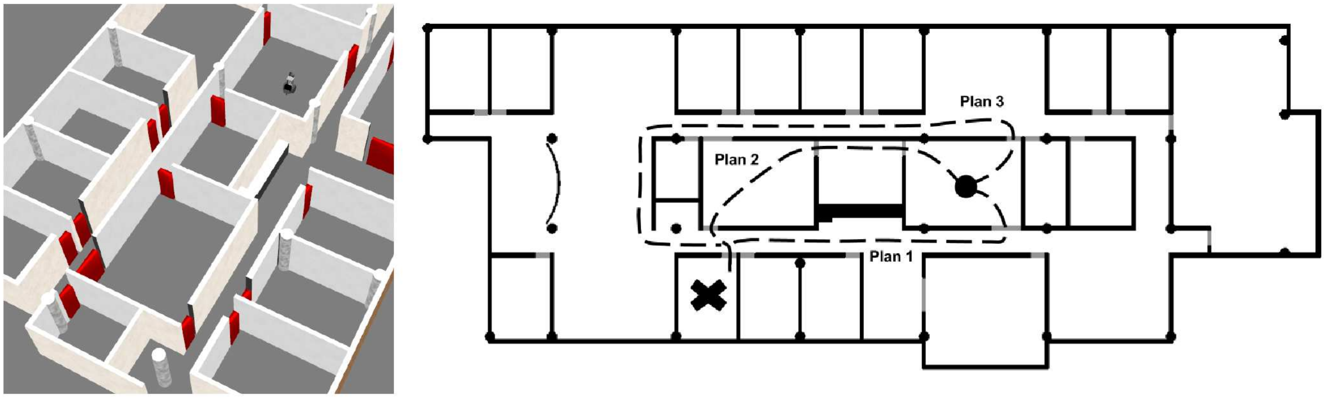
Planning with costs

In the previous section, the planner generates multiple plans of equal size 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, in the example in the previous section, when the robot plans to visit Carol to acquire Dan's location, the generated plan includes the robot exiting `lab1` through door `d4`. The planner also generated another plan of the same size where the robot could have exited through door `d5`, but that plan was not selected. If we see the layout of the example environment in Fig. 2, we can see that it is indeed faster to reach Carol's office `o3` through door `d5`. 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 the action. In this paper, for simplicity, we assume all actions apart from `approach` to have the following fixed costs:

```
cost(1,I):- sense(X,I), I<n.
cost(5,I):- gothrough(X,I), I<n.
cost(1,I):- greet(X,I), I<n.
cost(1,I):- opendoor(X,I), I<n.
```

We now define the cost for the `approach(X,I)` action, which depends on the physical location of the robot. In most



(a) Simulation in GAZEBO (b) Experiment floor plan

Figure 3: Final experiment domain which contains 20 rooms, 25 doors and multiple rooms with multiple doors. The circle marks the robot's start position, the cross marks the destination. The 3 plans evaluated in the results are also marked.

cases, fluents uniquely identify the physical location of the robot in the environment as the robot moves from one door to the next. We compute the cost of approaching door `X` from door `Y` in location `Z` as follows:

```
cost(@time(X,Y,Z),I):- approach(X,I),
  beside(Y,I),door(X),door(Y), at(Z,I), I<n.
```

In our domain, the physical location of the robot is uncertain at the start of an episode. It is impossible to estimate the true cost of approaching a door `X` unless the robot starts next to another door. When the robot starts next to a door, its physical location is expressed by a fluent and the true cost can be computed using the rule above. When it does not start next to a door, we assume the following fixed cost for approaching a door:

```
cost(10,I):- approach(X,I), {beside(Y,I)}0,
             I<n.
```

Finally, the following statement guides CLINGO to generate optimal plans in terms of the cumulative costs:

```
#minimize[cost(X,Y)=X@1].
```

As a consequence of planning with costs, the optimal plan is not necessarily the shortest one. Therefore, differently from the previous section, we do not repeatedly call CLINGO with incremental values of `n`. Rather, we call it once with `n` directly assigned to `max_len`. Using the optimization statement above, we guide CLINGO to find the optimal answer set, i.e. the optimal plan within a size of `max_len`.

Estimating costs through environment interactions

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*:

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

where k 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.

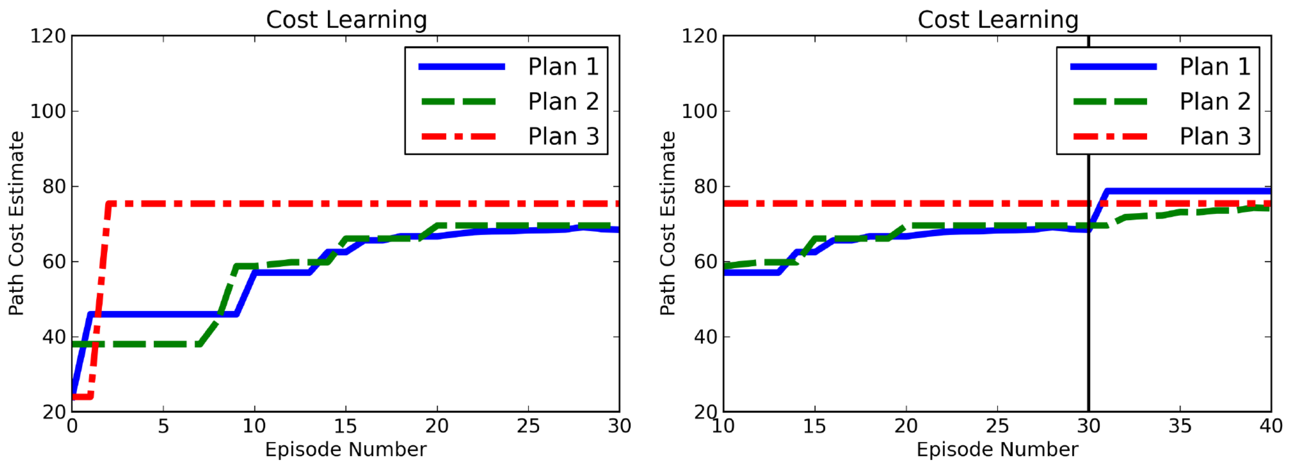
To apply this learning rule, we need estimates of all costs at episode 0. Since we want to explore a number of plans before choosing the lowest-cost one, we use the technique of *optimistic initialization* (Sutton and Barto 1998) and set all initial cost estimates to a value which is much less than the true cost. This causes the robot to underestimate the cost of a plan it has not explored enough, and the robot executes it to converge the action costs to the true values. As more and more plans are explored every episode, the costs of all relevant actions converge to their true values, and the planner settles on the true optimal plan. Since the cost of an action is independent of the task being performed, a robot can improve its cost estimates while executing different goals every episode.

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 higher 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 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 our approach of using ASP for plan cost minimization and cost learning in a simulated environment whose floor plan illustrated in Figure 3b. This environment has 20 rooms, 25 doors and 5 rooms with multiple doors, and a 3D simulation for this environment was implemented using GAZEBO (Koenig and Howard 2004), as depicted in Figure 3a. This environment uses all the same rules presented with the example, except that it uses a much larger corpus of rigid knowledge, which can be viewed online. A simulated differential drive robot capable of understanding the actions described in this paper moves around inside this simulation to complete goals. The robot's interface also transforms any simulated sensor readings into observable fluents as required by the planner to determine if replanning is required. Actions inside the simulator are not guaranteed to succeed. However, on the occasional failure, they do not fail repeatedly so that replanning can still be successful.

The experiments were carried out on a machine with a Quad-Core i7-3770 processor, where the processing was



(a) Cost estimation during the first 30 episodes (b) Cost estimation after discovering navigation delay in episode 30

Figure 4: The cost curves of three different plans in the environment during learning.

split between 3D simulation, visualization and the planner CLINGO. CLINGO is a monolithic system made up of two parts, the grounder GRINGO which generates variable-free programs which are then solved by the answer set solver CLASP. In these experiments, version 3.0.3 and version 2.1.4 of GRINGO and CLASP were used, respectively. CLINGO was run in parallel over 6 threads with `max_len` set to 15, and was allowed 1 minute of planning time after which the best available plan was selected for execution.

In this domain, there are a large number of correct plans, and the planning time can be prohibitively long. For instance, there are a number of plans that go into the rooms on the top left corner of the map, or at least approach the door to that room. We remove such plans from the search space of the planner by providing the following additional heuristics along with the goal:

- Do not approach a door if that door is the only door connecting the adjacent room to the goal of visiting Bob.

```
:- approach(Y,I), at(X,I), acc(X,Y,Z),
 1{indacc(Z,Y1,W)}1,inside(bob,W,I), Z!=W.
```

  Note that to formulate this constraint we use `indacc`, which is recursively defined in the rigid knowledge of the domain.

- Do not approach a door if the next action in the plan does not go through it. The formal definition of this heuristic has not been presented here due to its length, and can be viewed online.

These heuristics reduce the average time for finding the optimal plan at the start of each episode from 23 to 14.275 seconds. More importantly, they greatly reduce the number of episodes required to converge learning by explicitly identifying a number of plans as sub-optimal, which need not be evaluated.

We demonstrate cost learning through repeated episodes of a single-goal problem illustrated in Fig. 3b. In every episode, the robot starts at the location indicated by the circle and attempts to greet a person in the location indicated by the cross. At the end of each episode, the robot updates its estimates of action costs. Even with the heuristics mentioned above, there are still a large number of plans that can achieve this goal. For instance, there are plans that go through the seminar room and the lab as well, as without learned costs the planner has no idea that going through either of them does not shorten the distance to the goal. Although all these plans are tried out by the planner, we only present the costs for the three shortest plans after exiting each door in the initial room. Plan 1 and Plan 3 are the shortest plans to achieve the goal (7 steps), and Plan 1 is also the optimal plan. Plan 2 is almost as good as the optimal plan, but requires the robot to go through 4 doors, making it longer (13 steps). In this experiment, we only learn costs for the `approach` action.

Figure 4a shows the total cost of the three plans as the cost estimates for each individual action are improved starting from the optimistically initialized values. The cost of plan 3 is significantly higher than the other two, and therefore after trying the plan once in the second episode, it is not considered again. Since the other two plans are of similar cost, their execution is interleaved until the estimate of the shortest plan converges. In some episodes, plans apart from these three are executed and the value of these plans may not change. By episode 30, the cost estimates converge such that plan 1 is always selected.

In the second experiment we show how the system can

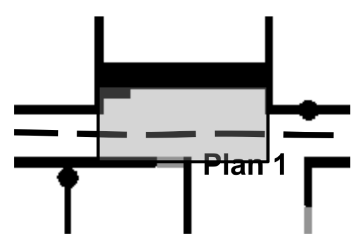


Figure 5: A delay in a portion of the map slows down robot navigation in the gray region shown above, and lengthens the time taken to complete Plan 1 at episode 30.

adapt when the environment changes. Let's assume that at episode 30, the robot starts experiencing a 20 second delay in the gray region of the map demarcated in Fig. 5, through which the optimal plan (Plan 1) passes. Consequently, the cost estimate changes and the robot switches to a different plan (Plan 2) which is now optimal, as illustrated over the next 10 episodes in the graph in Fig. 4b.

Conclusion

In this paper, we present an approach that uses ASP for robotic task planning that incorporates action costs to produce optimal plans. This approach also allows to plan with incomplete information, by acquiring missing information through human-robot interaction. Furthermore, by estimating costs from experience, planning can be adaptive to environmental changes.

Acknowledgments

The authors would like to thank Vladimir Lifschitz for many useful discussions during the seminal stages of this work. The authors would also like to thank the ROS and GAZEBO open-source communities for infrastructure used in this work.

A portion of this research has taken place in the Learning Agents Research Group (LARG) at the AI Laboratory, University of Texas at Austin. LARG research is supported in part by grants from the National Science Foundation (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, University of Texas at Austin.

References

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)*.

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)*.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence (AIJ). Presented at the International Joint Conferences on Artificial Intelligence (IJCAI)*.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Schneider, M. 2011a. Potassco: The Potsdam answer set solving collection. *AI Communications: The European Journal on Artificial Intelligence (AICOM)*.

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

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*.

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)*.

Jimnez, S.; Fernndez, 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)*.

Lin, F. 1995. Embracing causality in specifying the indirect effects of actions. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

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

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

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*.

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

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)*.