Learning Classical Planning Domains through Experience

Stephan Chang
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil

Abstract

One of the many applications of Machine Learning techniques to Automated Planning is for the task of knowledge acquisition. Knowledge acquisition is important because it affects the quality of a domain's description, and therefore the success of a planner in solving its problems. In this paper, we research an algorithm that acquires domain knowledge from observing actions examples that belong to the domain. We evaluate its performance by planning with the learned domain specifications and then give insights on future work.

1 Introduction

In this paper, we to address the question "How does learning happen?". We discuss a theory called "Learning from experience", or "Learning from the environment" , referring to past work in autonomous learning (Shen 1991), and propose an algorithm that is capable of knowledge acquisition in classical planning domains. The purpose of this project is the development of an autonomous learning technique, which uses Automated Planning as its underlying formalism, providing a common ground for benchmarking. Our goal is to use our algorithm to learn classical planning domains and use this to solve any problem instances within those domains. We evaluate our work by selecting a few domains and problems to train our learning program, which outputs the domain it has learned from its experience, and then attempting to solve different problem instances with this learned domain. From the planning perspective, this can be advantageous for knowledge acquisition, where we obtain a pre-encoding of an unknown domain. From the learning perspective, this allows us to understand how autonomous learning actually happens in the real world.

This paper is organised as follows. First, we define what learning from experience is in Section 2. Then, in Section 3, we describe an algorithm that learns from experience and how it can be integrated to classical planning, outlining the most important elements that need to be considered for a complete solution to our problem. Once the method is established and implemented, we proceed to make experiments to test the validity of our proposal, discussing them in Section 4. In Section 5, we conclude the paper, stating what was learned from the project and also giving some ideas of future work.

2 Learning from Experience

The process of "learning from experience" can be described as acquiring domain knowledge by generalising observations made within this domain. In learning from experience, an observer agent has access to a collection of actions executed by other agents, or by itself when it has exploratory abilities. As the agent makes observations, it studies the premises and conclusions of each action to reconstruct the domain by deducing its characteristics from observed data.

In planning, the observations are given in terms of states and actions. States are collections of logical statements that describe the world at a given time step, while actions are what agents can do to change the world's state. Therefore, in learning from experience, it is important to know the state the world was in at a given time step, what actions were executed at that time step, and to where those actions lead our world. In the end, the learning agent should have built a domain from a set of observations, and use this domain to solve a set of problems.

3 The Method

According to (Celorrio et al. 2012), there are four main topics of interest when combining learning with planning: knowledge representation, extraction of experience, the learning algorithm itself and the exploitation of learned knowledge. Knowledge representation is about how we represent the domain we are learning about. In extraction of experience we define how our training examples are collected, such as if they are obtained by the agent itself by exploring actions in the domain or if they are obtained from observing other agents acting in the domain. The next step, learning algorithm, is about how we use these extracted experiences to fill and refine our knowledge base. After accumulating knowledge, the agent needs to put it to the test, and evaluate whether it has learned enough or if there are still inconsistencies in the learned domain model. This last step is the exploitation of learned knowledge.

In our work, we use Planning Domain Definition Language (PDDL) for the task of Knowledge Representation, so that the output of our algorithm can be run directly on classical planners such as JavaGP and MauPlanner. To represent

training examples, we use tuples of the type $\langle S_i, a_i, S_{i+1}\rangle$, where $S_i = \{s_{i0}, s_{i1}, \ldots, s_{in}\}$ is a collection of predicates representing one state into the past; $a_i = \langle \alpha, p_0, p_1, \ldots, p_n\rangle$ is the action α executed at time step i with parameters $p_{0:n}$; S_{i+1}, is a collection of fluents similar to S_i, but representing the state of the world posterior to the execution of action a_i in state S_i.

Having defined how experiences are represented, we proceed to discuss the implemented algorithm. The algorithm we present can be run both online and offline, although our experiments are made with offline training only. In the case of offline training, we iterate through a set of training examples, while in online training we perform the following steps for each training example as they come. For each training example $\langle S_i, a_i, S_{i+1}\rangle$, we parse all predicates observed in both S_i and S_{i+1}, adding previously unobserved predicates in its ungrounded form to the domain definition. We give a unique name for each term in the predicate, so we can later map each term to a value and also avoid adding duplicate definitions. After adding any unknown predicates to our domain, we start processing actions.

In PDDL, action definitions have four components: the action's name, its parameters, preconditions and effects (both positive and negative effects go into the same list). One characteristic of classical planning with the PDDL formalism which helps with the task of learning preconditions and effects is that the predicates contained within these lists can only relate to objects referred to by the parameters of the action. So we can eliminate any predicates in S_i and S_{i+1} that are unrelated to any of the observed parameters. Suppose we have the action $move(x, from, to)$, and a set of preconditions $\{p(x), p(y), q(from, to)\}$. In this case, this means we can outright eliminate $p(y)$ from our preconditions, because the y element is unseen in this action's parameter list. We show this procedure in Algorithm 1 (lines 2 to 7).

Algorithm 1 The algorithm for inducing new actions from observation tuples.

1: **function** PARSEACTION($S_i, a_i = \langle \alpha, p_0, \ldots, p_n\rangle, S_{i+1}$)
2: **for each** $s = \langle id, l_0, \ldots, l_n\rangle \in S_i$ **do**
3: **if** $l_{0:n} \not\subseteq a_i$ **then**
4: $S_i \leftarrow S_i - s$
5: **for each** $s = \langle id, l_0, \ldots, l_n\rangle \in S_{i+1}$ **do**
6: **if** $l_{0:n} \not\subseteq a_i$ **then**
7: $S_{i+1} \leftarrow S_{i+1} - s$
8: $\phi \leftarrow S_i$ ▷ ϕ is the precondition.
9: $\epsilon^+ \leftarrow S_{i+1} - (S_i \cup S_{i+1})$ ▷ ϵ^+ are the positive effects.
10: $\epsilon^- \leftarrow S_i - (S_i \cup S_{i+1})$ ▷ ϵ^- are the negative effects.
11: $A \leftarrow \langle \alpha, p_{0:n}, \phi, \epsilon^+, \epsilon^-\rangle$ ▷ A is the induced action.
return A

Following the removal of irrelevant predicates, we select which predicates to add to the action's preconditions and effects. The selection process is similar for all lists, and is described in lines 8 to 10 of Algorithm 1. The action's precondition is the same set of predicates contained in the current state. To fill positive and negative effect lists, we need to take a different approach. We assume positive effects to be any predicates that appear in the posterior state that were not seen in the previous state. Likewise, we assume negative effects to be any predicates that are not seen in the posterior sense, but were seen in the previous state. Hence, the best way to implement this is using set theory with union and difference operations. Anything in the intersection of posterior and previous states is seen as a constant, and therefore is removed from effect lists.

The procedure we have just detailed covers the addition of unknown actions to the domain. In this work, we propose a method to use new observations of known actions to refine the model we have of them, which we show in Algorithm 2. As we know that all predicates contained in both precondition and effect lists have 100% chance of being observed when they are relevant to the action being executed, we simply compute the intersection between the current information and the new information. If the information is relevant to the action, it is sure to be present at all times, and if some predicate ceases to be seen between executions of the same action, then we know it is not a precondition for that action.

Algorithm 2 The procedure for refining known action models.

Require: α, the current action name
Require: $p_{0:n}$, the current action's parameters
Require: ϕ, the current action's preconditions
Require: ϵ^+, the current action's positive effects
Require: ϵ^-, the current action's negative effects
1: **function** MERGE($a_i = \langle \alpha, p_{0:n}, \phi', \epsilon^{+'}, \epsilon^{-'}\rangle$)
2: $\phi \leftarrow \psi \cap \phi'$
3: $\epsilon^+ \leftarrow \epsilon^+ \cap \epsilon^{+'}$
4: $\epsilon^- \leftarrow \epsilon^- \cap \epsilon^{-'}$

An interesting property of this approach is that, as repeated action executions are observed, we adjust the domain model to conform to these observations. Meaning that the model converges as more observations are made, and once converged, the model is able to solve problems for that domain. In turn, the problem of having enough training examples arises, as it is not possible to derive a complete domain with poor examples.

4 Experiments

In this section we describe our experimentation process to evaluate the potential of our solution. Initially, we modified the JavaGP planner to output not only the actions performed at each stage, but also the state prior to the action execution and the state posterior to the action execution. Because the examples are generated from the planning graph, our program learns at least how to solve that specific problem. For this same reason, however, the generated training examples may lack variety, therefore affecting the quality of the learned domain description. To alleviate this problem, we switched to MauPlanner, which executes an exhaustive search algorithm for planning, generating a greater volume of examples for training.

We run experiments on three classical planning domains: Hanoi, Blocks World and Ferry. Among these domains, we

select a few problems that serve as the training set, and others that serve as the evaluating set. We generate the training examples from running MauPlanner on the instances of the training set, using the output of this process as the input to our program. After running our program, we use the generated PDDL file to plan the instances in the evaluation set. We consider the learning task successful if all instances in the evaluation set are solved with similar performance to the original specification. As we will see, some domains are only partially learned, *i.e.* some of the instances in the evaluation set cannot be solved, in which case we must raise the number of training examples and re-create the domain. There is also the possibility that the learned domain will be able to solve problems, but with an inferior performance to that of the original domain. We provide our insights as we review each of the experiments.

Hanoi

The Hanoi experiment contains only 5 problem instances, of which we use the first one to generate the training set. This experiment is quite straightforward, and yields success on all evaluation problem instances. We observed a small inconsistency in the learned domain, in which there is an extra "smaller" predicate in the "move" action. Because this is a constant, however, problem solving was unaffected. We can see the comparison between learned and original versions of this action in Listings 1 and 2.

Listing 1: The learned *move* action with an extra precondition *(smaller ?x1 ?x0)*.

```
(:action move
    :parameters (?x0 ?x1 ?x2)
    :precondition (and (smaller ?x2 ?x0
        ) (on ?x0 ?x1) (clear ?x2) (
        clear ?x0) (smaller ?x1 ?x0))
    :effect (and (clear ?x1) (on ?x0 ?
        x2) (not (on ?x0 ?x1)) (not (
        clear ?x2))))
```

Listing 2: The original *move* action without the extra precondition.

```
(:action move
    :parameters (?disc ?from ?to)
        :precondition (and (smaller ?to
            ?disc) (on ?disc ?from) (
            clear ?disc) (clear ?to))
        :effect (and (clear ?from) (on
            ?disc ?to) (not (on ?disc ?
            from)) (not (clear ?to))))
```

We summarise our experiment's results in Table 1, which shows that training our domain with the first problem instance as the training set is enough to make it able to solve the remaining instances, and with performance similar to the original. This can be seen in columns "Time" and "Steps", in which we compare these metrics between planning with the learned domain and with the original domain specification.

Table 1: Results obtained from planning with both learned and original domain definitions for the Hanoi domain.

Examples	Pb	Time/Original(s)	Steps/Original
24	1	0.0002/0.0007	7/7
24	2	0.0010/0.0016	15/15
24	3	0.0056/0.0004	31/31
24	4	0.0278/0.0005	63/63
24	5	0.1334/0.2235	127/127

Blocks World

In the Blocks World domain we start with the training set containing the first two instances of the problem set. It is now that we begin to see the implications of lacking a fine variety of examples. The domain learned from poor examples may contain inconsistencies in the preconditions or effects of actions, making it harder or even impossible to find a solution for some problem instances. An example of this can be seen in Listings 3 and 4, where the *unstack* action in the learned domain contains an unnecessary precondition of the type *(onTable ?x1)*.

Listing 3: The learned *unstack* action definition in the Blocks World domain with 29 training examples.

```
(:action unstack
    :parameters (?x0 ?x1)
    :precondition (and (ontable ?x1) (
        clear ?x0) (on ?x0 ?x1))
    :effect (and (clear ?x1) (holding ?
        x0) (not (clear ?x0)) (not (on ?
        x0 ?x1))))
```

Listing 4: The original action definition in the Blocks World domain.

```
(:action unstack
  :parameters (?ob ?underob)
  :precondition (and (on ?ob ?underob)
      (clear ?ob))
  :effect (and (holding ?ob) (clear ?
      underob) (not (on ?ob ?underob)) (
      not (clear ?ob)))))
```

In Table 2, we summarise the experience of learning in the Blocks World domain. Up to problem 4, the domain learned from the training set is enough to achieve a planning performance similar to the one attained with the original domain definition. On problem 4, however, the inconsistencies in the learned domain prevent it from finding a proper solution. At this step, we insert a new instance to the training set, raising the number of training examples to 50, but still to no avail. Our insight is that none of the examples provided by solving problem instance 3 is distinct from the ones previously learned, *i.e.* the examples lacked diversity. To test this assumption, we tried a different approach to expand our training examples, in which we generated a partial solution for problem 4 (instead of generating the whole solution) and added it to the training set, raising the number of examples to 75. The result is the convergence of the domain, and the

successful planning of the fourth problem instance, which still had not been solved. We finish this round by using this domain to plan the remaining problems in the evaluation set, from which we obtain results similar to those obtained by using the original domain.

Table 2: Results obtained from planning with both learned and original domain definitions for the Blocks World domain.

Examples	Pb	Time/Original(s)	Steps/ Original
29	1	0.0000/0.000	2/2
29	2	0.0001/0.0001	4/4
29	3	0.0001/0.0002	3/3
29	4	0./0.0008	0/7
50	4	0./0.0008	0/7
75	4	0.0011/0.0009	7/7
75	5	0.01171/0.0103	5/5
75	6	0.1291/0.1371	6/6
75	7	2.5453/2.6251	7/7
75	8	45.5695/46.2668	8/8

Ferry Domain

The last domain we experiment with is the Ferry domain. From a body of 20 problem instances, we select one to be our training domain and 6 to be our evaluation domains. The training domain yields 15 training examples, from which our program is able to derivate a domain that is successful in resolving all problems in the evaluation set. In Table 3 we summarise the results we obtained from our experiments. We can see that all of the problems in the evaluation set were solved with the same performance as the original domain specification, even though the training set contained only a single problem instance. This enforces our hypothesis that diversity in training examples is more important than quantity.

Table 3: Results obtained from planning with both learned and original domain definitions for the Ferry domain.

Examples	Problem	Time/Original(s)	Steps/Original(s)
15	1	.0001/.0001	4/4
15	4	.0003/.0003	10/10
15	5	.0334/.0363	16/16
15	6	.0421/0.0431	15/15
15	7	.2232/.2559	15/15
15	19	21.5615/21.0984	26/26
15	20	39.3768/39.6934	21/21

One thing that stood out in this experiment is how the domain was learned by our algorithm. As shown in Listings 5 and 6, the *sail* action was learned with an extra precondition, *(not-eq ?x1 ?x0)*. This is because this predicate is a constant, much like its counterpart *(not-eq ?x0 ?x1)*, and was detected as present in all training examples. This does not seem to be an error, however. In fact, it makes sense that the predicate *(not-eq)* be symmetric, which calls our attention to the lack of this extra precondition in the original domain. While not groundbreaking, we believe this is fact is worth mentioning.

Listing 5: The learned *sail* action in the Ferry domain with 15 training examples.

```
(:action sail
    :parameters (?x0 ?x1)
    :precondition (and (not-eq ?x0 ?x1)
        (location ?x0) (location ?x1) (
       at-ferry ?x0) (not-eq ?x1 ?x0))
    :effect (and (at-ferry ?x1) (not (
       at-ferry ?x0))))
```

Listing 6: The original *sail* action in the Ferry domain.

```
(:action sail
       :parameters  (?from ?to)
       :precondition (and (not-eq ?from
           ?to)
                          (location ?
                              from) (
                              location ?
                              to) (
                              at-ferry ?
                              from))
       :effect (and  (at-ferry ?to)
                     (not (at-ferry ?
                         from))))
```

5 Conclusion

In this paper, we have discussed learning from experience and how we can apply it to learn classical planning domains. In deterministic and fully observable environments, it is possible to reconstruct a domain by observing the actions executed by other agents and using that information to induce the characteristics of the domain. It is important, however, that training examples have quality, rather than quantity, in which quality translates to diversity, *i.e.* that each example teaches something new to our algorithm.

There are a few venues for future work on learning from experience in planning domains. From the planning perspective, there are still some open problems in classical domains, such as learning negated preconditions and object types. Another interesting path to follow is in applying learning to HTN domains, where domain knowledge is even more important to planning performance. From the learning perspective, it is interesting to think of applying it in different types of environment, *e.g.* deterministic/non-deterministic, stochastic and partially observable/completely observable. Other approaches would also include distributed learning, in which agents are able to teach others by providing training examples to one another.

References

Celorrio, S. J.; de la Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *Knowledge Eng. Review* 27(4):433–467.

Shen, W. M. 1991. LIVE: An Architecture for Learning from the Environment. *SIGART Bull.* 2(4):151–155.