Learning actions success patterns from execution

Sergio Jiménez
Departamento de Informática.
Universidad Carlos III de Madrid.
Avda. de la Universidad, 30. Leganés (Madrid). Spain.
sjimenez@inf.uc3m.es

Abstract

Action policies allow to achieve robust plan executions in stochastic environments. Recent algorithms like LRTDP or LAO* efficiently find robust action policies but they require accurate specification of the dynamics of the environment. When addressing planning tasks in the real world these specifications are rarely available. Besides, learning them implies extensive exploration of the environment which can be immensely inefficient and even dangerous in some domains. As a result, off-the-shelf planners complemented with plan repair are frequently used to solve problems in these environments. In this paper, we discuss mechanisms to automatically provide off-the-self planners with information about the actions performance in the environment that improve the robustness the plans found. Specifically, this improvement is achieved by capturing patterns of situations in the environment that affects to the actions performance from observing plans executions.

Introduction

Recent planning applications like the control of Mars rovers, underwater vehicles or spacecrafts need to address planning tasks in stochastic environments. Two different approaches are mainly followed to solve these kind of planning problems:

- When the dynamics of the environment are known one can specify them as a Markov Decision Process (MDP). Algorithms like LRTDP (Bonet & Geffner 2003) or LAO* (Hansen & Zilberstein 2001) efficiently find good action policies by optimising a utility function, which gives preference to the best transitions of the MDP.
- When the dynamics of the environment are not available one can try to learn them. But learning from scratch the dynamics of a real environments is very complex. As it shown in (Pasula, Zettlemoyer, & Kaelbling 2004), even in simple toy worlds like the blocksworld, when actions are non-deterministic they may have innumerable outcomes thus the learning requires an extensive exploration of the world. So in practice, these problems are frequently addressed by off-the-self planners complemented with plan repairing (Fox *et al.* 2006) techniques.

This work is concerned with investigating how knowledge about the actions performance can be learned and used in off-the-self planners to find more robust plans. Specifically we study how to capture patterns of situations in the environment that affects to the actions success and how this information can be incorporated in a standard planning domain description.

Capturing the actions success

The process of learning the actions success consists of three phases: (1) experience gathering, (2) success patterns induction and (3) success patterns use. Figure 1 shows an overview of the integration of these three phases.

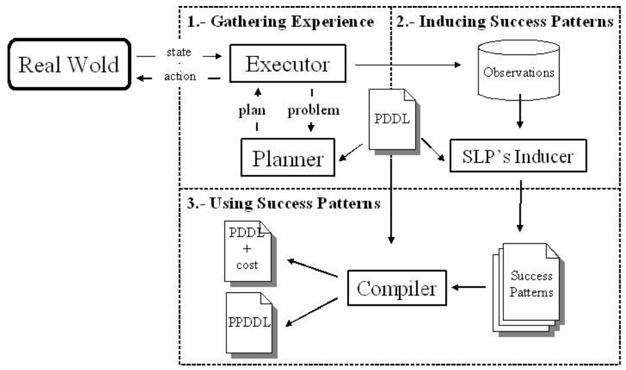

Figure 1: Process for capturing actions success patterns.

Phase 1: Experience gathering

In this phase the experience is gathered from observing plans execution. The inputs to this phase are a set of training examples and the STRIPS domain used by the planner to solve the training examples. Figure 2 shows the `pick-up` action from the STRIPS slippery-gripper domain. During this phase plans are generated and executed, and observations of the executions of actions in the environment are stored. The plans are generated by the heuristic planner FF (Hoffmann & Nebel 2001), but any off-the-shelf planner can be used. These plans are executed action by action and after each execution the corresponding observation is stored. The output of this phase is a set of observations of the form (s_n, a_i, s_{n+1}), where:

```
(:action pick-up
 :parameters (?b - block)
 :precondition (and (emptyhand)(clear ?b)(on-table ?b))
 :effect (and (holding ?b)
              (not (emptyhand))
              (not (clear ?b))
              (not (on-table ?b))))
```

Figure 2: Action `pick-up` from the STRIPS slippery-gripper domain.

- s_n is a conjunction of literals representing the facts holding before the action a_i execution;
- a_i is the action executed; and
- s_{n+1} is a conjunction of literals representing the facts that are observed to be holding after the action a_i execution.

When the state s_{n+1} of an execution observation does not correspond to the expected outcome of the executed action, the execution module calls the planner with the new state of the environment so the planner can replan to solve the problem in this new scenario. The execution of an action in an stochastic environment is performed by the simulator of the probabilistic track of the IPC5. This simulator allows us to generate environments where actions have stochastic behaviour using PPDDL1.0. The simulator maintains a representation of the current state of the environment that can be totally observed at any time, and that it is updated only when a new action is executed.

Phase 2: Success pattern induction

In this phase a relational machine learning mechanism is used to induce patterns of the success of the actions. The inputs of these phase are: the STRIPS domain theory and the gathered observations from previous phase. The output of the phase is a set of patterns of the success of the actions.

The patterns of the success of the actions are coded as Stochastic Logic Programs(SLP) (Muggleton 2000). SLP's are a generalisation of HMM's, stochastic context-free grammars, and direct Bayes nets. A SLP is a logic program with log-linear distributions associated to the clauses. The log-linear distribution provides a distribution over the variable bindings that allows SLP's to represent complex distributions. When the log-linear distributions of all the clauses of an SLP sharing the same head sums 1, the SLP is called *normalised SLP*. In a *normalised SLP*, the parameters of the log-linear distribution can be taken directly as probabilities. Figure 3 shows an example of a *normalised* SPL.

```
0.4 : s(X):- p(X),q(X).     0.3 : p(a).     0.2 : q(a).
0.6 : s(X):- q(X).          0.7 : p(b).     0.8 : q(b).
```

Figure 3: An example of a *normalised* SLP.

Inductive Logic Programming (ILP) have been successfully applied to SLP's learning (Muggleton 2000). ILP techniques are machine learning methods that induce logic programs that explain a given concept from examples represented as logic clauses. The ILP technique used [1] heuristically searches for the logic programs that explain as many positive examples of the target concept as possible and covering as less negative examples as possible. It is robust to noisy learning examples, allow incremental learning and support declarative background knowledge to guide the search. This technique is complemented with parameter estimation to obtain the parameters of the log-linear distribution associated to the induced logic programs.

Figure 4 shows the success patterns induced for the `pick-up` action from the slippery-gripper domain combining ILP and parameter estimation. The head of the clause represents the target concept (in the example the success of the action `pick-up`) and the body of the clause represents the set of predicates describing the conditions that make the target concept true. The parameters of the rule represent the probability of success of the action given that the body of the rule is true in the current state.

```
0.8:pick_up(Obs,Gripper,B1,B2):-not_wet(Obs,Gripper).
0.2:pick_up(Obs,Gripper,B1,B2):-wet(Obs,Gripper).
```

Figure 4: Success patterns induced for the `pick-up` action from the slippery-gripper domain.

Phase 3: Using the success patterns for robust planning

In this phase the success patterns are used to enhance the STRIPS domain theory. By the time being we have studied two solutions to improve the robustness of plans: (1) compiling the success patterns of an action into its probabilistic effects and (2) compiling the success patterns of an action into its cost value.

Solution 1: Compiling the success patterns into probabilistic effects. Each pattern p_j of the action a_i is converted into a conditional probabilistic effect of the action a_i: The condition of the effect is the body of the pattern p_j and the probability value is the parameter of the pattern p_j. Figure 5 shows how the two success patterns induced for the action `pick-up` are compiled into two probabilistic effects.

Solution 2: Compiling the success patterns into conditional cost. Each pattern p_j of the action a_i is compiled into a conditional cost of the action a_i: The body of the pattern is converted into the conditions of the cost and the parameter is translated into the cost value. Figure 6 shows the result of compiling the two success patterns induced for the `pick-up` action into two conditional costs.

The *fragility* of action a_i is computed as:

$$fragility(a_i) = -log(prob(a_i))$$

This expressions allows to transform the maximization of the product of the action success probabilities along the plan into a minimisation of the sum of the fragility costs. Searching for plans minimising this metric heuristically guides the planner towards robust solutions.

[1]http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph

```
(:action pick-up
 :parameters (?b - block)
 :precondition (and (emptyhand)(clear ?b)(on-table ?b))
 :effect (and (when (wet)
               (probabilistic
                  0.8 (and (holding ?b)
                           (not (emptyhand))
                           (not (clear ?b))
                           (not (on-table ?b)))))
              (when (not (wet)
              (probabilistic
                  0.2 (and (holding ?b)
                           (not (emptyhand))
                           (not (clear ?b))
                           (not (on-table ?b))))))))
```

Figure 5: Action `pick-up` updated with the probabilistic effects.

```
(:action pick-up
 :parameters (?b - block)
 :precondition (and (emptyhand)(clear ?b)(on-table ?b))
 :effect
  (and (holding ?b)
       (not (emptyhand))
       (not (clear ?b))
       (not (on-table ?b))
       (when (wet) (increase plan-fragility 0.2231))
       (when (not (wet)) (increase plan-fragility 1.6094))))
```

Figure 6: Action `pick-up` updated with the conditional cost.

Evaluating the use of success patterns

To evaluate the utility of the success patterns we have address 10 random problems of increasing difficulty from the probabilistic slippery-gripper domain. Every problem is solved 30 times, and we compared the solutions obtained by four different planning configurations measuring the average number of actions that each configuration needs to solve each problem:

- FF-Replan: FF (Hoffmann & Nebel 2001) plans with a STRIPS domain replanning on failure.
- FF-Replan with success knowledge: Metric-FF (Hoffmann 2003) plans with the numeric domain obtained from compiling the success patterns into fragility costs and replanning on failure.
- GPT with success knowledge. GPT (Bonet & Geffner 2004) plans with the probabilistic domain obtained from compiling the induced success patterns.
- GPT with complete knowledge. GPT (Bonet & Geffner 2004) plans with the exact probabilistic domain model. This configuration serves as a control to allow comparison.

Both configurations *FF-Replan with success knowledge* and *GPT with success knowledge* are the result of compiling the success patterns induced after solving 5 training problems. Figure 7 shows that both compilations of the success patterns allow to obtain more robust plans which solve the planning task using fewer actions.

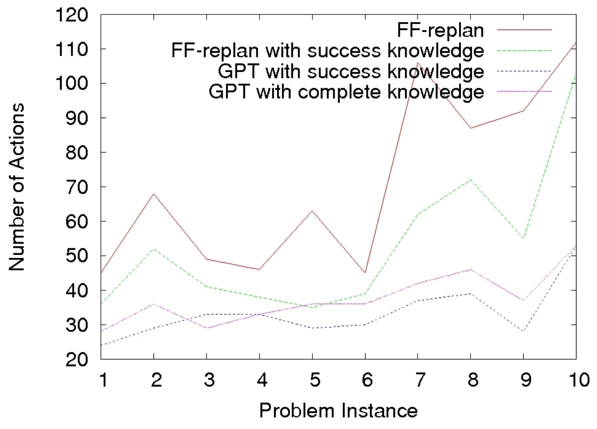

Figure 7: The obtained experimental results.

To date the learning mechanism have been implemented and two different ways of using the acquired patterns are been evaluated. The implementation of incrementally learning mechanisms to on-line incorporate the new learnt knowledge is still in progress.

References

Bonet, B., and Geffner, H. 2003. Labeled rtdp: Improving the convergence of real-time dynamic programming.

Bonet, B., and Geffner, H. 2004. mgpt: A probabilistic planner based on heuristic search. In *Proceedings of the IPC4, ICAPS-04*.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. *Proceedings of the 16th ICAPS* 193–202.

Hansen, E. A., and Zilberstein, S. 2001. LAO * : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129(1-2):35–62.

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

Hoffmann, J. 2003. The metric-FF planning system: Translating ignoring delete lists to numerical state variables. *JAIR* 20.

Muggleton, S. 2000. Learning stochastic logic programs. In *Proceedings of the AAAI2000 Workshop on Learning Statistical Models from Relational Data*.

Pasula, H.; Zettlemoyer, L.; and Kaelbling, L. 2004. Learning probabilistic relational planning rules. *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*.