Inducing non-deterministic actions behaviour to plan robustly in probabilistic domains

Sergio Jiménez, Fernando Fernández and **Daniel Borrajo**
Departamento de Informática
Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
sjimenez@inf.uc3m.es, ffernand@inf.uc3m.es, dborrajo@ia.uc3m.es

Abstract

In the probabilistic track of the last International Planning Competition two main approaches were used, Markov Decision Processes (Boutilier, Dean, & Hanks 1998) and decision-theoretic planning (Blythe 1999). Both approaches use a domain representation with an explicit definition of the probabilities of the actions effects. But when planning in realistic domains, most of the times, the non deterministic effects and the probabilities associated to them are unknown or hard to be obtained accurately. In this paper we present the LUCK architecture (Learning Uncertainty information as Control Knowledge). This architecture plans to solve problems in probabilistic domains using an initial deterministic domain representation. Then, it learns information about the success and the failure of the actions applying Inductive Logic Programming Techniques. And, finally, it uses this information to generate better plans (in terms of robustness) in the future.

Introduction

In this paper we present the LUCK architecture. This architecture solves problems in probabilistic domains planning from a deterministic representation of the domain and learning knowledge about the reasons that cause the execution of actions to be a success or a failure. This knowledge is translated into Control Knowledge so that the planner can reason about the uncertainty of the plans. The main two contributions of this paper are:

- Our system does not reason explicitly about a probabilistic description of the domain in the searching process. The knowledge related to the uncertainty of the world is modelled through Control Knowledge, which is separated from the domain model. So, in the search tree, there is no probabilistic knowledge.
- The probabilistic information of the domain is represented as Control Knowledge. This Control Knowledge is generated automatically from the experience using Inductive Learning Programming Techniques. Specifically we use the ALEPH system that induces theories that explain when an action will succeed or fail according to the current state.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

In the last International Planning Competition (IPC4[1]) practically all the planners that took part in the probabilistic track were solving MDPs based on planners. Among these planners there were only one using machine learning techniques to deal with probabilistic domains. This planner implements the ideas explained in (Fern, Yoon, & Givan 2004) to find policies specifying what action to take given a goal and a current state. And in the whole competition only the Probapop system, a conformant probabilistic planner (Onder, Whelan, & Li 2004), generates plans instead of policies given a probabilistic planning problem.

Another different approach to plan in non deterministic domain is planning based on a model checking approach (Kabanza, Barbeau, & St.-Denis 1997). This kind of systems is able to deal with complex goals as they use temporal model logic to describe them and don't need to know the probabilities of the effects of the actions. But they need a non-deterministic description of the domain, and again they try to find policies specifying what action to take given a goal and a current state.

These kinds of policies cannot easily be communicated to humans and typically is a difficult task to transfer this knowledge to other similar problems. In our approach, we don't learn policies but explanations of the causes that lie behind the success or failure of the actions. So the domain designer can get feedback information about the world dynamics from the execution of plans.

This fact also motivates the work (Cocora *et al.* 2006). In this paper is described how to generate Relational Markov Decision Processes (RMDPs) (Kersting, Otterlo, & Raedt 2004) and how to learn policies for these RMDPs using relational decision trees which are relational declarative representations of the policies. But this work is not precisely comparable with ours as it starts from a set of example plans, whereas our system starts from a deterministic domain description of the world.

There is previous work by Karen Haigh (Haigh & Veloso 1998) very related to ours, in which Control Knowledge is learned from the experience of executing actions to improve the planning process. In her work, the attributes that affect the execution of actions are known "a priori". In our work, we do not know "a priori" which attributes from the

[1] http://ipc.icaps-conference.org/

state are relevant with respect to the execution of actions. Reinforcement Learning approaches (Kaelbling, Littman, & More 1996) can also be considered as related work as they also repeat cycles of planning, acting and learning. However, since our approach is based on deliberative planning, it is able to reason with a richer representation of the domain. Thus, we can solve more flexibly problems with different goals and infinite number of potential states.

Instead of learning policies to achieve particular goals, Kaelbling et al. proposed to learn plan operators. In (Pasula, Zettlemoyer, & Kaelbling 2004) and in (Zettlemoyer, Pasula, & Kaelbling 2005), they described how to learn completely the actions model from examples. Our approach is different, since we initially have a deterministic representation of the domain and we only want to learn when the execution of an action is going to succeed or fail. From a planning point of view we believe that designing and maintaining a deterministic domain is a simpler task for the users. So, separation of domain knowledge from probabilistic knowledge can benefit in the process of generating applications, since declarative Control Knowledge can also be accessed.

The rest of the paper is organized as follows: first, we present the general architecture of the system and the planning, execution and learning processes. Next, we describe the experiments carried out to evaluate the architecture, and finally we discuss some conclusions.

The planning, acting and learning cycle

LUCK is an architecture that integrates planning, execution and learning. Figure 1 shows a high level view of its architecture. When LUCK faces a planning problem, it first proposes a plan to solve it, and then tries to execute the plan actions one by one. While LUCK executes actions it observes the results of these executions. When the execution of an action is a failure, LUCK plans to obtain a new plan that solves the problem from this current state. LUCK considers the execution of an action a failure when the new state resulting from the execution of this action is different from the state expected, according to its deterministic representation of the domain.

Initially, LUCK proposes plans to solve the first problem only taking into account the deterministic description of the domain. And, as it starts to observe the results of executing the actions in the real world, it will generate Control Knowledge that will guide it towards solutions that consider the uncertainty in the domain.

Planning

For the planning task we have used the non-linear backward chaining planner IPSS (Rodrguez-Moreno *et al.* 2004), based on PRODIGY4.0 (Veloso *et al.* 1995). The inputs to the planner are the usual ones in planning (domain theory and problem definition), plus declarative Control Knowledge, described as a set of Control Rules. The output of the planner, as we have used it in this paper, is a totally-ordered plan. The Control Rules act as domain dependent heuristics. They are one of the main reasons why we have used this planner, given that they provide a declarative representation of Control Knowledge.

The IPSS planning-reasoning cycle involves as 'decision points': choose a goal from the set of pending goals and subgoals; choose an operator to achieve the selected goal; choose the bindings to instantiate the chosen operator; and apply an instantiated operator whose preconditions are satisfied or continue subgoaling on another unsolved goal. The default decisions at all these decision points can be directed by Control Rules in order to guide the planner. In our approach, initially, the planner is executed without any Control Rules.

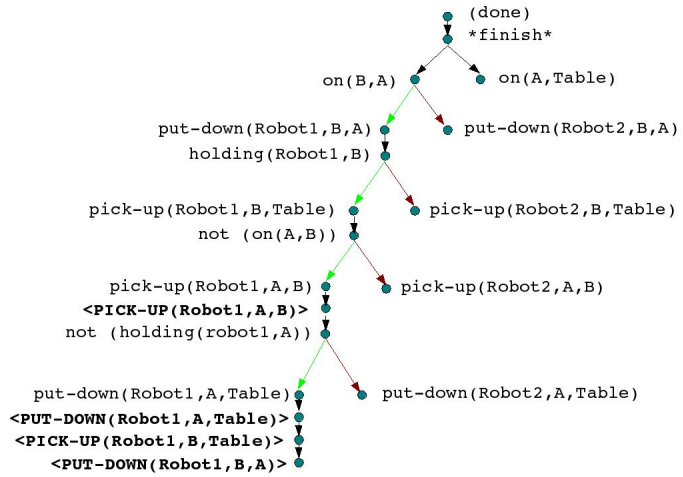

Figure 2: Example of a IPSS planner search tree.

Figure 2 shows a search tree where all the bindings decisions has been directed to prefer the Robot1. This tree belongs to the search process of the planner IPSS solving a problem consisting on reaching the goals `on(B, A)` and `on(A,table)` from a initial state described by the predicates `on(A, B)` and `on(B,table)`.

Actions execution

We simulate the execution of the actions in the non deterministic world. We use the simulator provided by the probabilistic track of the last International Planning Competition, IPC4[2], to evaluate probabilistic planners.

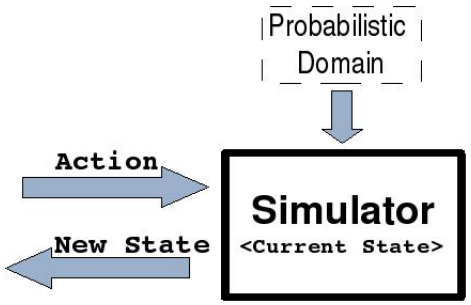

Figure 3: High level view of the simulator module.

[2]http://ipc.icaps-conference.org/

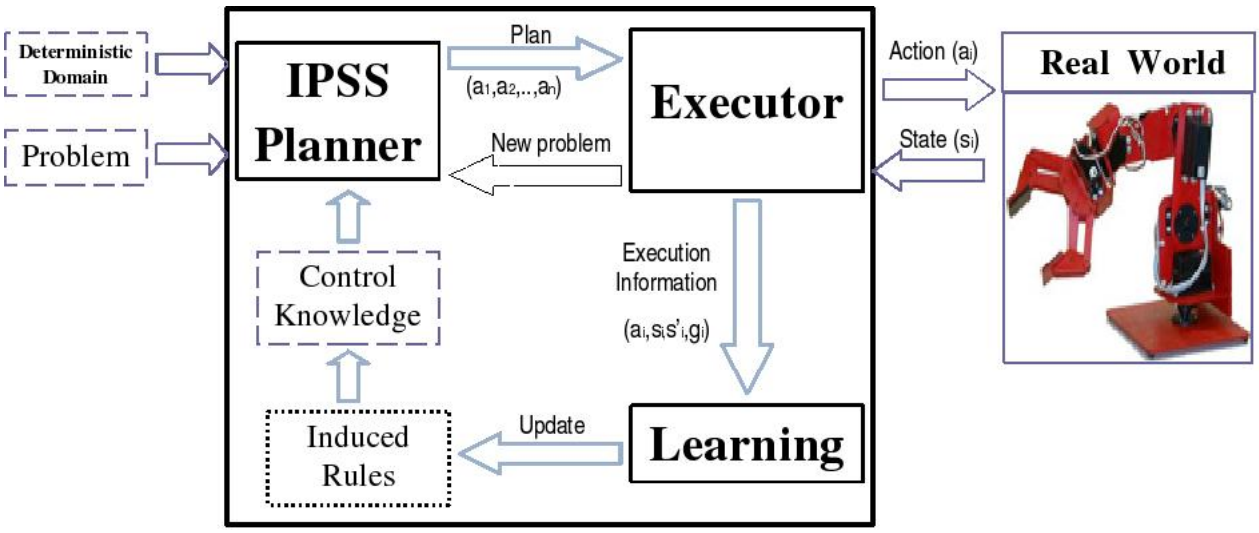

Figure 1: High level view of the LUCK planning-execution-learning architecture.

This simulator uses PPDDL1.0 (Younes & Littman 2004) to describe the world we want to simulate. This language allows us to describe actions with probabilistic and conditional effects. The simulator maintains a representation of the current state and updates it when an action is executed. Figure 3 shows the inputs and outputs of the simulator.

Learning from execution episodes

LUCK uses ILP (Inductive Learning Programming) techniques to analyze the data it obtains from executing actions. The examples used by the inductive learning techniques are tuples of the form: `(action, result, state)`, where `action` is the name of the executed action; `result` is the result of the action execution, that is `success` or `failure`, considering an action execution to be a failure when the new state caused by this action is different from the expected one according to the deterministic representation of the domain; and `state` is the current state when the action was executed. Figure 4 shows an execution step in a probabilistic blocksworld.

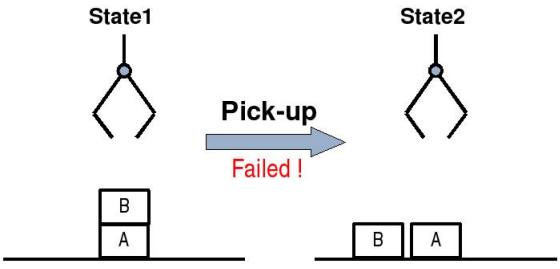

Figure 4: LUCK considers the execution of the action `pick-up-block-from(Robot1, B, A)` a failure.

For every different action of the domain, LUCK maintains the examples of success and failures of execution. From these examples, LUCK induces two different kinds of theories: theories about why the actions succeed and theories about why the actions fail.

To induce these theories, we use ALEPH [3] (A Learning Engine for Proposing Hypotheses). The ALEPH system is based on the Stephen Muggleton's ideas of inverse entailment (Muggleton 1995). This system proposes hypothesis, PROLOG programs, that cover a set of examples described using first order predicates. It can deal with noisy data (Dzeroski & Bratko 1992). And, it also can take as input PROLOG programs describing background knowledge that assists the induction process.

ALEPH receives three inputs:

- The Background Knowledge, which contains PROLOG clauses that encode information relevant to the domain. In our case, which are the types of the planning domain (blocks, robots, ...), which are the predicates that describe the examples (on, clear, ...), and which is the target concept to be learned (success-put-down-block-on, failure-put-down-block-on, ...).

  It also contains PROLOG clauses that encode the information obtained from the executions of actions, which are the learning examples. Figure 5 shows a learning example associated to the action `pick-up-block-from` for the state1 represented in Figure 4.
- The positive examples, that is a set of ground facts representing the positive learning examples of the concept to be learned.
- The negative examples which is a set of ground facts representing the negative learning examples of the concept to be learned.

The ground facts from the positive and negative examples have all the same appearance. They are PROLOG facts of the form

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

```
type-of-object(robot1,robot).
type-of-object(blockA,block).
type-of-object(blockB,block).
true-in-state-dirty(example1,robot1)
true-in-state-on-top-of(example1,blockB,blockA).
true-in-state-on-top-of(example1,blockA,table).
```

Figure 5: A learning example for the operator `pick-up-block-from`.

```
target-concept (example-id, parameters).
```

Where `target-concept` is the concept to be learned, `example-id` is a number to link the positive example to the associated leaning example, and `parameters` are the arguments of the operator. So, a ground fact from the positive or negative examples could be:

```
success-pick-up-block-from(example14,robot2,block4,table)
```

The output of ALEPH is a theory, a set of PROLOG clauses, that tries to cover as many positive learning examples as possible, covering the less number possible of negative examples . In LUCK this output represents a set of rules that will try to explain when an action succeeds and when it fails.

```
[Rule 1]
[Pos cover = 5 Neg cover = 27]
failure-pick-up-block-from(A,B,C,D) :-
    true-in-state-dirty(A,B).
```

Figure 6: Rule induced by AELPH to learn the concept of `failure` for the operator `pick-up-block-from`

These induced rules are composed of:

1. The number of positive and negative examples that the rules covers. That gives us an estimation of the rule credibility.
2. The head of the rule, that is target concept.
3. The body of the rule, predicates that describe when the target concept is true.

As an example, Figure 6 shows a rule that is part of a theory induced by ALEPH associated to the action `pick-up-block-from` to learn the concept of `failure`.

This rule means that when LUCK tries to `pick-up` a `block` using a `dirty robot-arm` from the `table` it is going to fail in about 15% of the times.

$$15\% \simeq Positives/(Positives + Negatives)$$

For each operator in the domain two sets of rules like the one in Figure 6 are generated: one set to explain the concept of success and other set to explain the concept of failure. Finally, LUCK is ready to use this information induced from the experience to tell the planner how to generate plans that consider the uncertainty.

Using the learned experience

LUCK uses the induced theories described in the previous section to automatically generate Control Rules that guide the planner in the decision points of the search tree. The *if-part* of the Control Rules is composed of the set of conditions that have to be satisfied in a given node of the search tree in order to fire that Control Rule. These conditions refer to aspects of the planner search process, such as what the current state is, in what goal the planner is working on, or what operator can achieve the current goal. The *then-part* of the rules describes what decision the planner should make in a 'decision point'.

```
(control-rule prefer-bindings-pick-up-block-from
 (IF
 (and
  (current-operator pick-up-block-from)
  (generate-best-binding-pick-up-block-from <best-binding>)))
 (THEN prefer bindings <best-binding>))
```

Figure 7: Example of control rule.

For the time being LUCK only generates Control Rules to guide the planner in decision points for choosing bindings. It generates automatically a Control Rule for every operator in the domain to choose the best bindings for that operator. Figure 7 shows an example of one of these Control Rules automatically generated by LUCK for the operator `pick-up-block-from`.

All these Control Rules have the same structure. They have two metapredicates:

- `(current-operator op)`. To fire the rule only when the current operator in the search tree is `op`.
- `(generate-best-binding-op <best-binding>)`. This metapredicate acts as a generator and it sets the variable `<best-binding>` to the best bindings for the current operator. The COMMONLISP code of this metapredicete is automatically generated taking into account the theories induced by ALEPH.Figure 8 shows the pseudocode of the metapredicate `generate-best-binding` for the operator `pick-up-block-from`.

generate-best-binding-pick-up-block-from (best-binding)

best-binding: The best bindings for the operator

B ← generate-possible-bindings(pick-up-block-from);
P ← Initialize-bindings-probabilities();
RS ← get-success-rules(pick-up-block-from);
RF ← get-failure-rules(pick-up-block-from);
For each possible binding b_i in B
  When b_i matches a success rule rs_j,
    increase probability of choosing b_i with reliability of rs_j
  When b_i matches a failure rule rf_j,
    decrease probability of choosing b_i with reliability of rf_j
best-binding ← choose-best-binding(B,P);
Return best-binding;

Figure 8: generate-best-binding pseudocode

First, the metapredicate generates all the possible bindings for the operator. Second, it associates a probability value to every possible binding. This probability is initially zero for all possible bindings. Then, it selects a binding, and tests if a success rule can be fired with it. If so, it increases the probability of choosing it considering the reliability of the rule. We are currently studying different ways of updating it. The initial method we are considering is just adding previous probability with the rules reliability. However, we will explore using no-regret techniques in order to update it. Then, it tests if a failure rule can be fired with this binding. If so, it decreases the probability of choosing the binding. This is done for all the possible bindings. And, finally, the function returns the best binding according to these probabilities.

Currently this function returns always the binding with the greater probability value. We are planning to use roulette mechanisms in the near future for selecting the best binding. But at this point we can consider using different exploration/exploitation strategies or even algorithms that also take into account the risk of executing actions.

Experimental Results

We have carried out experiments to evaluate the behaviour of the architecture using a modified version of the blocksworld domain from the probabilistic track of the IPC4. We have introduced three modifications:

- There are two robot arms (instead of one) to handle the blocks. Therefore, the operators `pick-up-block-from` and `put-down-bock-on` have another extra argument indicating the robot that carries out the action.
- There is a new predicate, indicating when a robot arm is dirty.
- When a robot is dirty, actions are going to fail 25% of the times. Obviously, the planner does not know it. It will have to learn it by executing actions.

The learned theories

In this version of the blocksworld domain LUCK has tried to solve ten *5-blocks* problems generated with the random problem generator provided by the probabilistic track of the IPC4. Then, it has generated theories about the success and failure of actions. Figure 9 shows the learned theories by LUCK for the operators `pick-up-block-from` and `put-down-block-on`.

- `Rule 1` means to LUCK that attempts to `pick-up` a block `C` from another block `D` succeeds 92% of the times.
- `Rule 2` means to LUCK that attempts to `pick-up` a block `C` from another block `D` with a dirty robot `B` fails 15% of the times.
- `Rule 3` means to LUCK that attempts to `put-down` a block `C` that is holded by the robot `B` on another block `D` succeeds 87% of the times.
- `Rule 4` means to LUCK that attempts to `put-down` a block `C` on another block `D` with a dirty robot `B` fails 25% of the times.

Induced Rules for `pick-up-block-from`

Success Rules

```
[Rule 1]
[Pos cover = 58 Neg cover = 5]
success-pick-up-block-from(A,B,C,D) :-
    true-in-state-on-top-of(A,C,D).
```

Failure Rules

```
[Rule 2]
[Pos cover = 5 Neg cover = 27]
failure-pick-up-block-from(A,B,C,D) :-
   true-in-state-dirty(A,B).
```

Induced Rules for `put-down-block-on`

Success Rules

```
[Rule 3]
[Pos cover = 42 Neg cover = 6]
success-put-down-block-on(A,B,C,D) :-
   true-in-state-holding(A,B,C).
```

Failure Rules

```
[Rule 4]
[Pos cover = 6 Neg cover = 18]
failure-put-down-block-on(A,B,C,D) :-
   true-in-state-dirty(A,B).
```

Figure 9: Learned theories for the operators `pick-up-block-from` and `put-down-block-on`.

For every operator two theories are automatically generated. A first one describing why an action succeeds and a second one describing when an action fails. In this example all the induced theories have just one rule.

As is done in our current work the estimation of the probability of the rules is not perfect, we explain how we plan to improve this estimation in the conclusions section. For example the induced probability for `Rule 2` should be 25% instead of 15%. When the amount of problems solved by LUCK is bigger, these values will be more accurate. For `Rule 1` and `Rule 3` these values depend on the number of times that these actions are tried with a dirty robot.

The point is that the estimation is not perfect but has information about why the execution of an action fails or succeeds. What allows LUCK to generate Control Knowledge to guide the planning module.

Measuring the quality improvement

To evaluate the worth of the learned theories we have solved a set of twenty-five *8-blocks* problems generated with the random problem generator provided by the probabilistic track of the IPC4. We have made LUCK to solve this twenty-five problem set with and without the acquired Control Knowledge described in the previous section. And we have measure two different magnitudes:

1. The length of the plan executed to solve a problem.
2. The number of failed actions. That is the number of replanning process.

As the success or failure of the actions is non deterministic, to obtain reliable values we have solved 15 times every problem and we have extracted the average values. Table 1 shows the obtained experimental results.

In all the problems, except in problem 12 and in problem 19, the number of failed actions is less or equal planning with the induced Control Knowledge than planning without it. In problem 12 and in problem 19 these values are practically the same (they differ just a little because we are testing in a probabilistic domain). So we can state that planning using the induced Control Knowledge makes LUCK finding more robust plans.

As the plans found using the Control Knowledge are more robust, less replanning process has been needed so the length of the executed plans is also shorter or equal than planning without the induced Control Knowledge.

And also, as the induced Control Knowledge acts as heuristics to the LUCK planning module, it makes LUCK to find solution in problems that couldn't be solved before within a time bound of 30 seconds (problem2, problem8, problem15, problem17, problem24).

Planning Time

To evaluate how the use of the Control Knowledge affects to the planning process we have measured the time that takes the planner to solve the twenty-five problem set. We have solved 15 times every problem and we have extract the average values. Table 2 shows the obtained experimental results.

Problem	Time	Time using CK
Problem1	0.37	0.68
Problem2	Unsolved	0.7
Problem3	0.28	0.61
Problem4	0.77	0.39
Problem5	0.21	0.42
Problem6	2.31	0.53
Problem7	0.73	0.38
Problem8	Unsolved	3.66
Problem9	0.42	1.72
Problem10	2.59	1.5
Problem11	0.36	1.59
Problem12	0.16	0.52
Problem13	0.4	0.46
Problem14	0.36	1.15
Problem15	Unsolved	2.96
Problem16	0.79	2.37
Problem17	Unsolved	1.8
Problem18	0.25	0.95
Problem19	0.78	1.33
Problem20	0.47	0.88
Problem21	0.36	2.1
Problem22	0.81	1.57
Problem23	0.2	1.7
Problem24	Unsolved	2.04
Problem25	1.61	2.705

Table 2: Experimental planning times for a set of twenty-five
8-blocks problems.

On one hand when planning with the Control Knowledge solutions are more robust, less actions fail and less replanning processes are needed so it takes less time to solve a problem. But on the other hand when the number of failures is practically the same planning with and without the Control Knowledge, the time that takes LUCK to solve a problem is a little bit bigger using Control Knowledge. Because planning using Control Knowledge implies making some extra computations in the planner search process that cost time.

Conclusions

In this paper, we present the LUCK architecture for acting in domains with uncertainty. The LUCK system acquires automatically information about the behaviour of the actions and acts according to plans that are obtained using this information.

Initially LUCK only needs a deterministic description of the action model since it handles the uncertainty learning declarative control rules that modify its default deterministic behaviour. We have designed this approach basically for three main reasons:

1. Defining a probabilistic domain for realistic problems is not an easy task. Usually the non deterministic effects of the actions and the probabilities associated to them are unknown or difficult to predict. Our approach automatically learns information that deals with this kind of uncertanity.
2. Leaning completely a domain theory without any kind of bias for a realistic problem is also a hard task (Pasula, Zettlemoyer, & Kaelbling 2004).
3. The induced declarative Control Rules give information understandable for a human about the causes that lie behind the success or failure of the actions

Experimental results reflects that LUCK is able to learn the reasons that causes the success and failure of the actions of a simple probabilistic domain. This information is kept separated from the action theory, represented as Control Knowledge and it is used to generate Control Rules that guide successfully the planning process towards solutions of a better quality.

In this paper the quality of the plans has been interpreted as the robustness of its actions. And the final goal lie in finding solutions to problems using the most robust actions in the minimum amount of steps. As it is done in the IPC4. So the learning process has been focused on the concepts of the failure or success of actions execution. An interesting extension for future efforts is working in domains where some other attribute of the actions has to be learned. For example, the time that takes an action to finish its execution or the cost of executing an action in a given domain.

Our current work lie in using symbolic statistical techniques to estimate the probabilities of the induced rules in an accurately way. Precisely we are working with the programming language for symbolic-statistical modelling PRISM[4]. PRISM perform a Maximum Likelihood estimation of the program parameters from incomplete data by

[4]http://sato-www.cs.titech.ac.jp/prism/

Problem	Failures	Failures using CK	Plan Length	Plan Length using CK
Problem1	1	1	40	40
Problem2	Unsolved	1	Unsolved	38
Problem3	1	1	30	30
Problem4	7.35	1	27.85	16
Problem5	1	1	24	24
Problem6	14.42	1	60.28	30
Problem7	5.71	1	28.75	20
Problem8	Unsolved	1	Unsolved	118
Problem9	1	1	46	46
Problem10	14.5	1	61.57	34
Problem11	1	1	38	38
Problem12	1.35	1.42	12.92	13
Problem13	4.24	1	13	10
Problem14	1	1	36	36
Problem15	Unsolved	1.71	Unsolved	72.71
Problem16	1	1	76	76
Problem17	Unsolved	1	Unsolved	52
Problem18	1	1	20	20
Problem19	1.85	2.07	29.71	29.85
Problem20	4.28	1	16.78	14
Problem21	1	1	40	40
Problem22	6	1	28.92	20
Problem23	1	1	22	22
Problem24	Unsolved	1	Unsolved	32
Problem25	6	2.07	45.5	36.64

Table 1: Experimental plan quality measures for a set of twenty-five *8-blocks* problems.

the Expectation-Maximization algorithm (Sato & Kameya 2001).

At the same time we are also working in how we can use the information learned from the experience not only to help the LUCK planner module choosing the correct bindings but also choosing the correct goals. Therefore, improving the quality of the solutions.

As the IPSS planner is not a fast planner, in the near future we plan to study how Control Knowledge can be learned to guide the search process of more efficient planners such as the heuristic planner FF (Hoffmann 2001) and thus compare our system to the *state-of-the-art planners* that take part in the probabilistic track of the International Planning Competition.

Acknowledgements

This work has been partially supported by the Spanish MEC project TIN2005-08945-C06-05 and regional CAM-UC3M project UC3M-INF-05-016.

References

Blythe, J. 1999. Decision-theoretic planning. *AI Magazine, Summer*.

Boutilier, C.; Dean, T.; and Hanks, S. 1998. Planning under uncertainty: structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*.

Cocora, A.; Kersting, K.; Plagemann, C.; Burgard, W.; and Raedt, L. D. 2006. Learning relational navigation policies. *Kunstliche Intelligenz*.

Dzeroski, and Bratko, I. 1992. Handling noise in inductive logic programming. *Workshop on Inductive Logic Programming, ICOT-TM-1182, Inst. for New Gen Comput Technology, Japan*.

Fern, A.; Yoon, S.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks (draft). *Proceedings of the International Conference in Automated Planning and Scheduling*.

Haigh, K. Z., and Veloso, M. M. 1998. Planning, execution and learning in a robotic agent. *AIPS* 120–127.

Hoffmann, J. 2001. Ff:the fast forward planning system. *AI Magazine, 22(3)* 57–62.

Kabanza, F.; Barbeau, M.; and St.-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95(1):67–11.

Kaelbling, L. P.; Littman, M.; and More, A. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*.

Kersting, K.; Otterlo, M. V.; and Raedt, L. D. 2004. Bellman goes relational. In *In Proceedings of the Twenty-First International Conference on Machine Learning (ICML-04)*.

Muggleton, S. 1995. Inverse entailment and Progol. *New*

Generation Computing, Special issue on Inductive Logic Programming 13(3-4):245–286.

Onder, N.; Whelan, G. C.; and Li, L. 2004. Probapop: Probabilistic partial-order planning. *Proceedings of the International Conference in Autommated Planning and Scheduling*.

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

Rodrguez-Moreno, M. D.; Borrajo, D.; Cesta, A.; and Meziat, D. 2004. An ai tool for scheduling satellite nominal operations. *AI Magazine*.

Sato, T., and Kameya, Y. 2001. Parameter learning of logic programs for symbolicstatistical modeling. *Journal of Artificial Intelligence Research* 391–454.

Veloso, M.; Carbonell, J.; Prez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI* 7:81–120.

Younes, H. L. S., and Littman, M. L. 2004. Ppddl1.0: An extension to pddl for expressing planning domains with probabilistic effects. *Technical Report CMU-CS-04-167, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.*

Zettlemoyer, L.; Pasula, H.; and Kaelbling, L. 2005. Learning planning rules in noisy stochastic worlds. *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*.