Deduction and Exploratory Assessment of Partial Plans

Jacek Malec and **Sławomir Nowaczyk**
Jacek.Malec@cs.lth.se and Slawomir.Nowaczyk@cs.lth.se
Department of Computer Science, Lund University,
Box 118, 221 00 Lund, Sweden

Abstract

In this paper we present a preliminary investigation of rational agents who, aware of their own limited mental resources, use learning to augment their reasoning. In our approach an agent creates and deductively reasons about possible plans of actions, but — aware of the fact that finding complete plans is in many cases intractable — it executes partial plans which look promising. By doing so, it can acquire new knowledge from results of performed actions, which allows it to plan further into the future in a more effective way.

We describe a possible application of Inductive Logic Programming to learn which of such partial plans are most likely to lead to reaching the goal. We also discuss how one can use ILP framework for generalising partial plans, thus allowing an agent to discover, after a number of episodes, a complete plan — or at least a good approximation of it.

1 Introduction

The basic idea of this project is to investigate a methodology for developing rational agents — both virtual and physical ones — that would be able to learn from experience, becoming more efficient at solving their tasks. A rational agent is expected to use deductive reasoning in order to take advantage of whatever domain knowledge it has been provided with. Besides that, it should perform inductive learning to benefit from experience it has gathered, correcting missing or inaccurate parts of that knowledge. Finally, it must acknowledge the fact that both reasoning and acting takes time, and try to balance those activities in a reasonable way.

In this paper we present how such rational agents can deal with planning in domains where complexity makes finding complete solutions intractable. Clearly, in many domains (especially those that are, at least from agent's point of view, nondeterministic) it is not realistic to expect an agent to be able to find a total plan which solves a problem at hand. Therefore, we investigate how an agent can create and reason about *partial plans*. By that we mean plans which bring it somewhat closer to achieving the goal, while still being simple and short enough to be computable in reasonable time. Currently we mainly focus on plans which allow an agent to acquire additional knowledge about the world.

By executing such "information-providing" partial plans, an agent can greatly simplify further planning process — it no longer needs to take into account the vast number of possible situations which will be inconsistent with newly observed state of the world. Thus, it can proceed further in a more effective way, by devoting its computational resources to more relevant issues.

We believe that the research field of planning has currently matured enough that it is time to explore new, more ambitious settings, in order to bring artificial agents closer to what humans are capable of. Our goal is to create an agent that is able to function in an adversary environment which it can only partially observe and which it only partially understands. Moreover, the agent is supposed to face a large number of episodes, learning from its mistakes and improving its efficiency.

We will base our examples on a simple game of Wumpus, a well-known test bed for intelligent agents, which is straightforward enough to properly illustrate our approach. In its basic form, the game takes place on a square board through which an agent is allowed to move. One square is inhabited by the Wumpus. Agent's goal is to kill the monster by shooting an arrow onto the square it occupies, while avoiding getting eaten by the monster. Luckily, Wumpus is a smelly creature, so the player always knows if the monster is on one of the squares adjacent to his current position — but unfortunately, not on which one. We leave the exact details of whether and how fast Wumpus can move open for now, since we will vary it in order to illustrate different ideas.

The main problem in the game of Wumpus is to learn the position of the monster. In order to plan for achieving this objective, an agent needs to be able to reason about its own knowledge and about how will it change as a result of performing various actions. Thus, the logic it utilises in its reasoning needs to strongly support epistemic concepts. At the same time, a notion of time-awareness is necessary, as we require our agent to consciously balance planning and acting.

To accommodate those requirements, we employ a variant of Active Logic [Elgot-Drapkin *et al.*, 1999] as the agent's underlying reasoning apparatus. This logic was designed for non-omniscient agents and has mechanisms for dealing with uncertain and contradictory knowledge. We believe it is a good reasoning technique for versatile agents, as it has

been successfully applied to several different problems — including some in which planning plays a very prominent role [Purang *et al.*, 1999].

The domain of Wumpus game has one more interesting feature, namely that the interesting behaviour of the agent consists of two phases. First, it has to gather some information ("Where is the Wumpus?") and, after that, it needs to exploit this knowledge ("How to get rid of it from there?"). Since this knowledge only becomes available during plan execution, not while agent is creating the plan, it needs to make its choice of actions depend on the previous observations of the world. Therefore, it has to create, reason about and execute conditional plans. Currently we have chosen a simple, straightforward way of representing conditional actions, although quite a few more advanced formalisms can be found in the literature [Russell and Norvig, 2003].

To summarise, our agent will create several different plans and reason about usefulness of each one — including what knowledge can be acquired by executing it. Further, it will judge whether it is more beneficial to immediately begin executing one of those plans or rather to continue deliberation. In other words, the agent will be performing on-line planning, interleaving it with plan execution. Moreover, we expect it to live much longer than any single planning episode lasts, so it should generalise each solution it finds. In particular, the agent needs to extract domain-dependent control knowledge and use it when solving subsequent, similar problem instances. Finally, it will have to be able to handle nonstationary, adversary environment, to cooperate with others in multi-agent setting and to plan for goals more complex than simple reachability properties (such as temporally extended goals and restoration goals).

All of the features mentioned above have been extensively studied in the planning literature, including ideas how to integrate various combinations of them — and we will discuss some of this work through this paper. However, to the best of our knowledge, nobody has yet attempted to merge all, or even most, of those features together in one, consistent framework.

This work is divided in the following way: the next section presents the architecture of our agent, describing the important modules and their functions, as well as how they interact. Sections 3–5 provide more detailed overview of each module separately. In Section 6 we briefly present some of the most relevant work done by other researchers. We conclude with summary and several ideas for further work.

2 Architecture

The architecture of our agent, presented in Figure 1, consists of three main elements. First of them is the Deductor, which performs deductive reasoning about world, actions and their consequences. Its main aim is to generate plans applicable in current situation. Furthermore, it predicts — at least as far as agent's past experience and imperfect domain knowledge allows — effects each of those plans will have, including what new knowledge can be acquired.

The second component is the Actor, which chooses and executes plans created by Deductor. It is also responsible for observing the world and introducing effects of actions — and, potentially, other changes in the environment — into agent's knowledge base. It is important to note that Actor determines *when* to stop deliberation and start execution of the chosen plan.

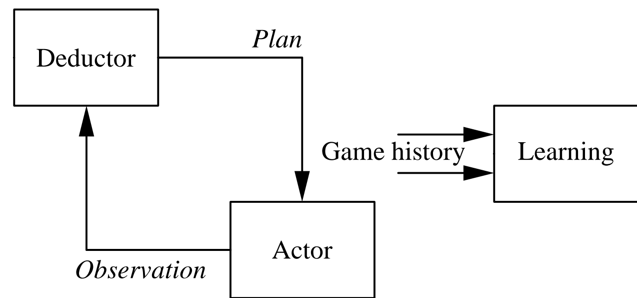

Figure 1: The architecture of the system.

These two modules form the core of the agent. By creating and executing a sequence of partial plans our agent moves progressively closer and closer to its goal, until it reaches a point where a winning plan can be directly created by Deductor, and its correctness can be proven.

However, success depends on whether the chosen partial plans are indeed moving an agent *closer* to the solution. Since agent's knowledge is incomplete and moreover it does not have enough resources to fully utilise the knowledge it possesses, there is — in principle — no guarantee that it will be so. In particular, if an Actor makes a mistake, the chosen plan may lead to loosing the game.

This is the reason for including the third module in our architecture. After the game is over, regardless of whether the agent has won or lost, learning system attempts to inductively generalise experience it has gathered — attempting to improve Deductor's and Actor's performance. We intend to use the learned information to fill gaps in the domain knowledge, to figure out generally interesting reasoning directions, to discover relevant subgoals and, finally, to more efficiently choose the best partial plan.

In principle, learning could take place at any time, but we do not currently see much benefit of learning in the middle of the game. Our variant of Wumpus game is simple enough that a single episode does not last very long, and there is some useful information that is only available to an agent after the game is finished — information which can be very valuable during learning.

3 Deductor

In order to present Deductor we begin with a description of the chosen knowledge representation formalism. Next we introduce those concepts from Active Logic which are necessary for understanding the rest of this text. We then present how the conditional actions are incorporated in our framework, and finally we illustrate how the three elements are combined for creating (partial) plans.

3.1 Knowledge representation

The language used by Deductor is the First Order Logic (FOL) augmented with Situation Calculus mechanisms for describing action and change. Within a given situation, knowledge is expressed using standard FOL. In particular, we do not put any limitations on the expressiveness of the language, as some mechanisms we later employ would invalidate benefits of restricting ourselves to languages such as Horn clauses or description logics. Predicate K describes knowledge of the agent, e.g.,

$$K[\, smell(a) \leftrightarrow \exists_x (Wumpus(x) \wedge Neigh(a, x)) \,]$$

meaning: *agent knows that it smells on exactly those squares which neighbour Wumpus' position*. The predicate K may be nested, although it is seldom useful. We use standard reification mechanism for putting formulae as parameters of the K predicate.

The next step is to introduce action and change representation. We use the well-known Situation Calculus approach, introducing predicates $Holds(situation, formula)$ to denote that the *formula* holds in *situation* and $Informs(action, groundedwff)$ to denote that *action* provides information whether *groundedwff* holds. We also introduce function $Result(situation, action)$, which returns the set of situations resulting from applying *action* in *situation*.

Another important concept in our formalism is a plan, which is a sequence of actions. Plans may be subject to concatenation operation. In every place where this might matter (in particular at argument list of the K predicate) we introduce two additional parameters. We denote by them, respectively, the set of situations and the set of plans to be executed (starting from those situations) in order to make the third argument true. So, actually, the formula shown above should look as follows:

$$K[\{s\}, \{p\},\ smell(a) \leftrightarrow \exists_x (Wumpus(x) \wedge Neigh(a, x)) \,]$$

meaning: *in a situation s agent knows that if it executes plan p then it smells on exactly those squares which neighbour Wumpus' position*.

This particular formula is true regardless of the chosen s and p (it is an universal law), a fact which we can denote, for example, by $\mathbb{S}$ (set of all situations) and either $\emptyset$ (empty plan) or $\mathbb{P}$ (the set of all plans). Still, there are many interesting formulae — like ones in the form "$Wumpus(x)$" — which are true *only* for specific s and p.

Please observe that the main notion our agent reasons about is its own knowledge about the world. Similar idea was introduced in [Petrick and Bacchus, 2004], where authors investigate how various actions and observations of their effects modify agent's belief state. They describe how such modifications can be propagated backwards and forwards through the state history: as the agent gains new knowledge, it can infer that various statements *did* hold in past states of the world, even if it did not know it then. Authors also show how such propagation can be used to deal with temporally extended and restoration goals.

3.2 Active Logic

Active Logic (AL) is intended to describe the deduction as an ongoing process, instead of characterising just some infinite, fixed-point consequence relation. To this end, it annotates every formula with a time-stamp (usually an integer) of when it was first derived, and bookkeeps the reasoning process by incrementing the label with every application of an inference rule. E.g.,

$$\frac{i:\quad a, a \rightarrow b}{i+1:\quad b}$$

Additional features, available in AL and important for this work, include the Now predicate, true only during current time point (i.e., "$i : Now(j)$" is true for all $i = j$, but false for all $i \neq j$) and the *observation function*, which delivers axioms that are valid since a specific time-point. It is used to model agent acquiring new knowledge from the environment. This way the reasoning process may refer, via Now, to the current (absolute or relative) time and conclude whether it has passed a deadline or not. It can also describe change that is not a result of performing any action — thus lifting two important limitations present in the classical situation calculus.

3.3 Conditional plans

The conditional plans we consider consist of a concatenation of classical and conditional actions, where each conditional action may be described as $(predicate\ ?\ action_1 : action_2)$, meaning that $action_1$ will be executed if $predicate$ holds, and $action_2$ will be executed otherwise. We consider the possibility of introducing a more complex structure of conditions (like *while* loops), but within this application simple conditionals will suffice.

This type of conditional actions introduces a high branching factor in case of longer plans, but this effect is unavoidable at some level of consideration and will not be further discussed here. It has received some attention in the works by other authors (see [Russell and Norvig, 2003] for extended bibliography).

For a well-developed discussion of conditional partial plans and interleaving planning and execution see for example [Bertoli *et al.*, 2004], where authors introduce notion of *progressive plan* — intuitively, one that provably moves the agent closer to the goal. They also present an algorithm for finding such plans in a nondeterministic but fully known domain and prove that it is guaranteed to find a solution if one exists.

A somewhat similar, very interesting idea was pursued in [Nyblom, 2005], where author uses classical planner to plan for "optimistic" case, where an agent can choose the most favourable outcome of each non-deterministic action. From such an optimistic plan it is then possible, using knowledge of probabilities of each action outcome, to generate more realistic plans by updating relative costs of optimistic actions.

3.4 Reasoning about plans

Finally, the representation language needs to be augmented with reasoning capabilities. It is done using a set of rather natural, although not quite trivial, inference rules. Their presentation, however, is outside the scope of this paper. Using

those rules, the Deductor may conclude, from the example formula shown earlier, that

$$\forall_x\, K[\mathbb{S}, \mathbb{P},\, \neg smell(a) \wedge Neigh(a,x)\,] \leftrightarrow \\ K[Result(s,p), \emptyset,\, \neg Wumpus(x)\,]$$

i.e., that *if it doesn't smell in position a then the agent will know that there is no Wumpus on any of its neighbour positions*. This may be further used for creating a useful plan of actions given that the agent currently is, or has been before, in position a.

One of the reasons we have chosen symbolic representation of plans, as opposed to a policy (an assignment of value to each state–action pair) is that we intend to deal with other types of goals than just reachability ones. For a discussion of possibilities and rationalisation of why such goals are interesting, see for example [Bertoli *et al.*, 2003], where authors present a solution for planning with goals described in Computational Tree Logic. This formalism allows to express goals of the kind "value *a* will never be changed", "*a* will be restored to its original value" or "value of *a* after time *t* will always be *b*" etc.

Furthermore, one of our ideas is to extend the solution presented in this paper to the case of multi-agent cooperative planning, where benefits of symbolic plan representation are even more clear.

To summarise, the agent uses the formalism presented in this section in order to deductively develop plans. Given the complexity of the domain and vastly branching proof procedure (currently it can only perform forward chaining) the created plans are usually partial, i.e. they lead to some intermediate states of the game, where the final outcome is not yet decided.

4 Actor

The Actor module supervises the deduction process and breaks it at selected moments, e.g., when it notices a particularly interesting plan or when it decides that sufficiently long time has been spent on planning. It then *evaluates* existing partial plans and executes the best one of them. The evaluation process is crucial here, and we expect the subsequent learning process to greatly contribute to its improvement. In the beginning, the choice may be done at random, or some simple heuristic may be used. After execution of partial plan, a new situation is reached and the Actor lets the Deductor create another set of possible plans.

This is repeated as many times as needed, until the game episode is either won or lost. Losing the game clearly identifies bad choices on the part of the Actor and leads to an update of the evaluation function.

Winning the game also yields feedback that may be used for improving this function, but it also provides a possibility to (re)construct a complete plan, i.e. one which originates from the initial situation and ends in a winning state. If such a plan can be found, it may be subsequently used to immediately solve any problem instance for which it is applicable. Moreover, even if such plan is not applicable, an Actor can use it when evaluating other plans found by the Deductor. Those which have similar structure to the successful one are more likely to lead to the goal.

In other words, the intention is for Actor to acquire generalised knowledge of the domain, which can be used to guide an agent in more promising directions.

In a sense this is similar to ideas discussed in [Fern *et al.*, 2004], where authors use Markov Decision Process to represent planning domains and approximate policy iteration as means of learning agent's behaviour. They use long random walks to create progressively harder goals, thus bootstrapping the agent in its learning of domain-dependent control knowledge.

5 Learning

As we mentioned earlier, our agent will be presented with large number of tasks to solve. Therefore, upon finishing each game episode, the events (actions, observations and the result) are fed into a learning module. This module attempts to generalise this information and provide guidelines for Actor and Deductor to improve their performance. In this paper we will mainly investigate the learning module from Actor's perspective, as using ILP framework to evaluate quality of partial plans is, to the best of our knowledge, a novel idea. In further work we also intend to improve domain knowledge and to identify interesting reasoning directions, but those later ideas are — while definitely interesting and non-trivial — mainly a matter of integrating the already available techniques.

5.1 Goal of learning

The first task we would like our learning module to address is how an Actor is to choose which one of the plans being considered by Deductor should it execute. Clearly, the longer it allows planning phase to proceed, the better plans will it get to choose from, and the more information about consequences of each plan will be known. On the other hand, more of the deduction effort will be wasted by considering potential situations which will not take place in this particular game.

At some point, however, an Actor must choose one plan, from those created by Deductor, for immediate execution. Some of those plans are better than others — but it cannot be determined exactly and with full confidence until those plans extend to the terminal state of the game. And for problems we intend to tackle, that is intractable — agent's computational resources do not suffice to *completely* solve problems we are interested in. Therefore, the Actor needs some heuristic method of evaluating quality of partial plans and of comparing them.

There is quite a bit of knowledge that domain experts could provide — but our aim is to have a solution which does not *require* such experts. At the same time, if people familiar with particular domain are available, the agent should take advantage of whatever information they can provide. Therefore, Inductive Logic Programming appears to fit our needs quite well: it uses background knowledge when it is available, but can also solve problems when it is not.

It is important to keep in mind that our agent has a dual aim, very akin to the exploration and exploitation dilemma, well-studied in reinforcement learning and related research

areas. On one hand, it wants to win the current game, but at the same time it needs to learn as much general knowledge as possible — in order to improve its performance at subsequent tasks.

5.2 Choosing plans

There are clearly many features which can distinguish between good and bad plans. And with sufficiently rich history of game episodes, it is possible to learn this distinction. In the simplest case the agent can start with Actor randomly choosing plans for execution. After a couple of games — some of which will be won but, likely, many will be lost — it will have enough experience to learn some useful rules.

The main problem is that most work on ILP, as well as on Machine Learning in general, has been dealing with the problem of *classification*, while what we need is rather *evaluation*. There is no predefined set of classes into which plans should be assigned. What our agent needs is a way to choose the *best* one of them.

Still, in order to be able to take advantage of the vast amount of research done in the Inductive Logic Programming framework, in the first step we recast our problems as a classification one. In particular, we attempt to distinguish plans that leading to losing the game from all the others. In our initial architecture this part is relatively easy — we assume that the Deductor has perfect knowledge of consequences of execution of each plan, so it can deduce (for some plans p) a fact "$K[\{s\}, \{p\}, \neg die]$".

A separate question is whether an Actor can *learn* to choose only plans for which "$K[\neg die]$" has been deduced. After all, not every plan for which such fact cannot be proven actually *does* lead to losing every game.

Moreover, it is worth noting that if the Wumpus is allowed to move, there exist plans which do not lead to agent's death, but which do lead to states where winning it is no longer possible — for example, if an agent gets stuck in a corner with Wumpus blocking its way out. It may be difficult for an agent to notice and learn that the mistake has been made in the previous step, not in the one when the agent was killed.

5.3 Application of ILP

From the above analysis it becomes clear than the notion of *positive* and *negative* examples, as used in ILP algorithms, is not quite appropriate for what we would like to express in our framework. What they correspond to, informally, are conditions that are both *necessary* and *sufficient* — while we are mainly interested those that are sufficient.

An interesting line of research, which possibly could be useful in our case, was presented in [Gretton and Thiebaux, 2004], where authors attempt to deductively generate domain-specific hypothesis language which is as simple as possible, and yet expressive enough to represent all the necessary concepts in a particular domain. This language is then used by inductive learning algorithm to create generalised policies from solutions of small problem instances.

Let us assume that we restrict ourselves to dividing plans into two classes: those that can lead to agent's death and those that cannot. Each partial plan executed at some previous game can be seen as a single example. First issue we need to deal with is which example belongs to which class. It is easy to note that some plans — namely those that in agent's experience *do* lead to losing the game — are definitely examples of bad plans. However, not every plan which does not cause the agent to die is, indeed, a *good* plan. What is more, not every plan that leads to *winning* a game is a good one. An agent executing a dangerous plan might have just gotten lucky, if in a particular episode Wumpus was in favourable position.

Therefore, if we want ILP algorithm to learn the concept of bad plans, we do have a set of positive examples, and a set of examples for which we do not — at least not immediately — know their affiliation. We have decided to use PROGOL as a learning algorithm. The standard version is presented in [Muggleton, 1995] and can be described here, in a somewhat simplified manner, by the following steps:

1. Select an example to be generalised. If no more examples exist, stop.
2. Construct the most specific clause, within provided language restrictions, which entails selected example. This is called the "bottom clause".
3. Find, by searching for some subset of the literals in the bottom clause, more general clauses. Choose one with the best "score".
4. Add the clause found in the previous step to the current theory, and remove all clauses made redundant. It is worth noting that the best clause may make clauses other than the examples redundant. Move back to Step 1.

In our case we can define as positive examples those plans which lead — or can be *proven* to *possibly* lead — to agent's death. On the other hand, those plans which can be proven to *never* lead to the agent's death are treated as negative examples. We are working on ways to utilise other plans in some way, those for which neither of the above assertions can be proven (within reasonable time) — right now we simply exclude them from learning.

With the definitions as above, we can use standard ILP algorithm, be it PROGOL or almost any other, to have Actor learn to choose only *non-losing* plans for execution.

However, this is only a beginning. After all, it is not quite enough not to die, as an agent which moves in circles, without exploring the world, clearly does not get eaten by the Wumpus — but it never wins either. On the other end of the spectrum, the feature "plan which kills the Wumpus" is clearly non-operational.

Hopefully, we will be able to report more details on practical applicability of the ideas described above when our implementation is finished and we have run some experiments.

5.4 Further ideas

One very promising idea seems to be exploring the epistemic quality of plans. An agent should pursue those plans which provide it with the most important knowledge. Clearly, in the Wumpus domain *important* is directly linked with monster's true position — or at least that is what human players consider it to be. Therefore, as a next step, we can redefine *bad* plans as those that lead to the agent's death or do not provide any

interesting knowledge. Again, we can use one of many ILP algorithms to learn such concepts.

Another very general and important way of expressing distinction between good and bad partial plans, and one we feel can lead to very good results, is related to discovering relevant subgoals and landmarks in the plans, akin to the work done in [Hoffmann *et al.*, 2004].

The problem is that those ideas require more domain knowledge than we are comfortable with. For example, what we would like to have is an agent figuring out that "position of Wumpus" is important just from the definition of the rules and goals of the game. In principle, it appears to be possible — it is not difficult to deduce that knowing Wumpus' position suffices for winning the game (the plan to win once Wumpus' position is known is simple and can be found easily). However, it is not clear how to combine such reasoning with learning as expressed above. It is our understanding that some modifications to the learning algorithm will be required.

To summarise, it is easy (for a human) to see some general rules distinguishing good plans from bad ones. For example, a plan for which an agent doesn't know that it will not lose the game is a bad plan. Such knowledge can be easily provided by domain expert and most ILP algorithms are ready to use it. Interesting question, however, is whether and how can this knowledge be discovered by an agent itself.

One way would be to try something along the lines of research presented in [Walker *et al.*, 2004], where authors randomly sample a large number of relational features and evaluate them on small problems. The idea is that features found to work satisfactory on such sample problems should also describe larger problems sufficiently well.

6 Related Work

Combination of planning and learning is an area of active research, in addition to the extensive amount of work being done separately in those respective areas. However, most of the related work we are aware of is devoted to either using state-of-the-art learning in a rather limited planning framework, or to using limited learning in a more complex planning setup. Comparisons of the two areas are also relatively common, while the true, nontrivial combination will apparently require much more investigation. Since we believe it to be very promising, this paper is aiming at attracting attention to this line of research.

The first to mention is [Dietterich and Flann, 1995], which presented results establishing conceptual similarities between explanation-based learning and reinforcement learning. In particular, they discussed how EBL can be used to learn action strategies and provided important theoretical results concerning its applicability to this aim.

There has been significant amount of work done in learning about what actions to take in a particular situation. One notable example is [Khardon, 1999], where author showed important theoretical results about PAC-learnability of action strategies in various models.

In [Moyle, 2002] author discussed a more practical approach to learning Event Calculus programs using Theory Completion. He used extraction-case abduction and the ALECTO system in order to simultaneously learn two mutually related predicates ($Initiates$ and $Terminates$) from positive-only observations.

Recently, [Könik and Laird, 2004] developed a system which is able to learn low-level actions and plans from goal hierarchies and action examples provided by experts, within the SOAR architecture.

The work mentioned above focuses primarily on learning how to act, without focusing on reaching conclusions in a deductive way. In a sense, the results are somewhat more similar to the reactive-like behaviour than to classical planning system, with important similarities to the reinforcement learning and related techniques. In case of large search spaces this approach may not be as effective as a suitable combination of learning and deduction. Therefore, some effort have been devoted to searching for a suitable combination.

One attempt to escape the trap of large search space has been presented in [Džeroski *et al.*, 2001], where relational abstractions are used to substantially reduce cardinality of search space. Still, this new space is subjected to reinforcement learning, not to a symbolic planning system.

A conceptually similar idea, but where relational representation is actually being learned via behaviour cloning techniques, is presented in [Morales, 2004].

Outside the domain of planning, there is a lot of interesting research being done in the learning paradigm.

Recently, [Colton and Muggleton, 2003] showed several ideas about how to learn interesting facts about the world, as opposed to learning a description of a predefined concept. A somewhat similar result, more specifically related to planning, has been presented in [Fern *et al.*, 2004], where the system learns domain-dependent control knowledge beneficial in planning tasks.

From another point of view, [Khardon and Roth, 1995] presented a framework of learning done "specifically for the purpose of reasoning with the learned knowledge" — an interesting early attempt to move away from the *learning to classify* paradigm.

Yet another track of research focuses on (deductive) planning, taking into account incompleteness of agent's knowledge and uncertainty about the world. Conditional plans, generalised policies, conformant plans and universal plans are the terms used by various researchers [Cimatti *et al.*, 2004; Bertoli *et al.*, 2004] to denote in principle the same idea: generating a plan which is "prepared" for all possible reactions of the environment. This approach has much in common with control theory, as observed in [Bonet and Geffner, 2001] or earlier in [Dean and Wellman, 1991]. We are not aware of any such research that would attempt to integrate learning.

As can be seen, many of the ideas we investigate in this paper have been analysed previously, but an attempt to merge them into a single, consistent framework has not yet been made.

7 Conclusions and Further Work

The work presented here is more a discussion of an interesting track of research than a report on some concrete results. However, we think that this idea is important and promising

enough to be subjected to wider discussion, and therefore we have decided to present it in this forum.

We have introduced an agent architecture facilitating resource-aware deductive planning interwoven with plan execution and supported by inductive, life-long learning. The particular deduction mechanism used is based on Active Logic, in order to incorporate time-awareness into the deduction itself. The plans created in deductive way are conditional, taking into account possible results of future actions, in particular information-gathering ones.

The learning mechanism employed is based on PROGOL, although in principle any standard ILP algorithm would be suitable as well. Learning is expected to provide an evaluation of the current deductive knowledge in order to improve the agent's performance in the long run.

We are at the moment working on implementation of the system and expect to be able to report results of first experiments at the time of the workshop.

In the future we intend to continue this work in the following directions:

- Discovering subgoals and subplans. It seems that one of the most useful capacities of humans problem solving is the ability to divide a complex problem into subproblems and then to solve each of them separately before combining their solutions into a global one. We would like to force our agent to discover this possibility. In our example domain a useful subgoal/subproblem could be "First, find a place where it smells."
- Discovering general rules which Deductor will be able to use later on. An example of such a rule might be "Don't shoot if you don't know Wumpus' position". It seems that availability of such rules can save a substantial amount of work for Deductor, if it can establish early on that some plans would not be usable.
- Generalisation of plans. A clear advantage would be to reuse a valid plan in a different context. As long as the context does not differ substantially, this operation should lead to fast solution of a problem similar to one solved in the past.
- Capability of handling imperfect knowledge. The current setup assumes complete domain knowledge, while in many situations this assumption might be violated (e.g., the agent might not know that the Wumpus actually can move). The system should allow the agent to learn domain knowledge, if possible, to complete its understanding of the environment.
- Last, but not least, allow interaction with a user. Domain experts might be an invaluable source of knowledge that the agent should be able to exploit, if possible. For example, to better adjust tradeoff between spending time on deduction and induction, the agent could be guided by an external observer (the user) providing a feedback about its performance.

The list above does not cover all the possible further investigations and extensions of the proposed system; it is just a biased presentation of the authors' own interests and judgements.

Acknowledgements

The authors are grateful to the anonymous reviewers for their very thorough comments and suggestions, which lead to a substantial improvement of this paper.

References

Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, and Paolo Traverso. A framework for planning with extended goals under partial observability. In *International Conference on Automated Planning and Scheduling*, pages 215–225, 2003.

Piergiorgio Bertoli, Alessandro Cimatti, and Paolo Traverso. Interleaving execution and planning for nondeterministic, partially observable domains. In *European Conference on Artificial Intelligence*, pages 657–661, 2004.

Blai Bonet and Hector Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

Alessandro Cimatti, Marco Roveri, and Piergiorgio Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206, 2004.

Simon Colton and Stephen Muggleton. ILP for mathematical discovery. In *13th International Conference on Inductive Logic Programming*, 2003.

Thomas Dean and Michael P. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

Thomas G. Dietterich and Nicholas S. Flann. Explanation-based learning and reinforcement learning: A unified view. In *International Conference on Machine Learning*, pages 176–184, 1995.

Saso Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine Learning*, 43(1/2):7–52, 2001.

Jennifer Elgot-Drapkin, Sarit Kraus, Michael Miller, Madhura Nirkhe, and Donald Perlis. Active logics: A unified formal approach to episodic reasoning. Technical Report CS-TR-4072, University of Maryland, 1999.

Alan Fern, SungWook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *International Conference on Automated Planning and Scheduling*, 2004.

Charles Gretton and Sylvie Thiebaux. Exploiting first-order regression in inductive policy selection. In *Conference on Uncertainty in Artificial Intelligence*, 2004.

Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Khardon and Roth. Learning to reason with a restricted view. In *Proceedings of the Workshop on Computational Learning Theory, Morgan Kaufmann Publishers*, 1995.

Roni Khardon. Learning to take actions. *Machine Learning*, 35:57–90, 1999.

Tolga Könik and John Laird. Learning goal hierarchies from structured observations and expert annotations. In *14th International Conference on Inductive Logic Programming*, 2004.

Eduardo P. Morales. Relational state abstraction for reinforcement learning. In *Proceedings of the ICML'04 Workshop on Relational Reinforcement Learning*, 2004.

Steve Moyle. Using theory completion to learn a robot navigation control program. In *12th International Conference on Inductive Logic Programming*, 2002.

Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

Per Nyblom. Handling uncertainty by interleaving cost-aware classical planning with execution. In *Swedish AI Society Workshop*, 2005.

Ronald P. A. Petrick and Fahiem Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 2–11, 2004.

Khemdut Purang, Darsana Purushothaman, David Traum, Carl Andersen, and Donald Perlis. Practical reasoning and plan execution with active logic. In *Proceedings of the IJCAI-99 Workshop on Practical Reasoning and Rationality*, pages 30–38, 1999.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in AI, 2nd edition, 2003.

Trevor Walker, Jude Shavlik, and Richard Maclin. Relational reinforcement learning via sampling the space of first-order conjunctive features. In *In working notes of ICML-04 Workshop on Relational Reinforcement Learning*, 2004.