Strategy Learning for Reasoning Agents

Hendrik Skubch and Michael Thielscher

Department of Computer Science
Technische Universität Dresden
Germany
hendrik.skubch@inf.tu-dresden.de, mit@inf.tu-dresden.de

Abstract. We present a method for knowledge-based agents to learn strategies. Using techniques of inductive logic programming, strategies are learned in two steps: A given example set is first generalized into an overly general theory, which then gets refined. We show how a learning agent can exploit background knowledge of its actions and environment in order to restrict the hypothesis space, which enables the learning of complex logic program clauses. This is a first step toward the long term goal of adaptive, reasoning agents capable of changing their behavior when appropriate.

1 Introduction

Endowing agents with the cognitive capability of reasoning is a major research topic of Artificial Intelligence [1]. The high-level control of a reasoning agent comprises two parts: a background theory which contains knowledge about actions and their effects, and a goal-oriented strategy according to which the agent reasons and acts. Existing programming methods, such as GOLOG [2] or FLUX [3], require the programmer to provide both the background theory for the underlying domain and strategies in view of specific goals. Learning techniques have recently been applied to let agents find out the effects of their actions from experiments [4, 5], but the learning of goal directed strategies on top of this has not yet been considered.

In this paper, we present a method to learn strategy programs from examples using Inductive Logic Programming (ILP). As the underlying action formalism we use FLUX, a logic programming method for the design of intelligent agents, based on the action formalism of the fluent calculus [6]. One of the key advantages of combining reasoning about actions with learning is that agents can use their background knowledge to considerably restrict the hypothesis space. Thus it can become possible to learn rather complex clauses including negated conjunctions. Strategies are learned in two steps: First, the given examples are generalized based on the notion of Least General Generalization of [7]. The resulting, overly general theory is then refined to obtain a strategy program that is sound and complete wrt. the given example set.

In the next section, we briefly recapitulate the basics and notations of the agent programming method FLUX. In Section 3, we define the general hypothesis

space for FLUX strategies and show how the background theory of a reasoning agent can be used to restrict the search space. In Section 4, we present a method for constructing overly general strategies from examples, and in Section 5 we then explain how these theories are corrected by specialization. In Section 6, we present and discuss experimental results. We conclude in Section 7.

2 FLUX

The fluent calculus [6] is an axiomatic approach for representing and reasoning about actions and change. The basic notion is that of a state and its atomic components, the so-called fluents. The fundamental predicate $holds(F, Z)$ is used to express that fluent F is true in state Z. Actions are specified in the fluent calculus by precondition and effect axioms.

Based on logic programming, FLUX is a method for the design of agents that reason logically about their actions. The background theory BK of a FLUX agent consists of a kernel program encoding the foundational axioms of the fluent calculus, along with domain-dependent knowledge in form of domain constraints, precondition axioms, and state update axioms. Here, we focus on a simplified variant of FLUX in which agents have complete state knowledge, called *Special FLUX* in the remainder of this paper.

On top of the background theory BK, the behavior of FLUX agents are given by logic programs that describe acting strategies. The agents use a state as their mental model of the world, on the basis of which they decide which action to take. As they move along, the agents constantly update their world model to reflect changes they have effected and sensor information they have acquired.

As an example, Figure 1 depicts a FLUX control program for a simple elevator originally formulated in GOLOG [2]. The states in this domain are composed of the three fluents `cur_floor(N)`, `on(M)` and `opened` meaning that the elevator is at floor `N`, the button for floor `M` has been activated, and the door is open, respectively. The elevator can perform the actions `up(N)`, `down(N)` of going up (respectively, down) to floor `N`; `turnoff(N)` of turning off the button at floor `N`; and `open`, `close` of opening and closing the door.

3 Hypothesis Space

The hypothesis space is the space of all programs the Inductive Inference Machine (IIM) might consider as a solution to a learning problem. A strategy for a Special FLUX agent is a logic program selecting an action to be executed in each state. Thereby the strategy relates states to actions. We use a single, recursive predicate to express strategies:

```
loop(Z):- strategy(Z,A) ->
          (A \= stop, execute(A,Z,Z2), loop(Z2); true).
```

Here, the predicate `strategy/2` selects the action to be executed. Given that the program does not change for any Special FLUX problem, this is the only predicate to be learned.

```
main(Z) :- serve_a_floor(Z,Z1) -> main(Z1) ; park(Z).

serve_a_floor(Z,Z1) :- holds(cur_floor(N),Z), holds(on(M),Z),
                       \+ (holds(on(M1),Z), closer(M1,M,N)),
                       serve(M,Z,Z1).

serve(M,Z,Z4) :- go_floor(M,Z,Z1), execute(open,Z1,Z2),
                 execute(turn_off(M),Z2,Z3), execute(close,Z3,Z4).

go_floor(M,Z,Z1) :- poss(up(M),Z)   -> execute(up(M),Z,Z1) ;
                    poss(down(M),Z) -> execute(down(M),Z,Z1) ;
                    Z1=Z.

park(Z) :- execute(down(1),Z,Z1), execute(open,Z1,_).

closer(M1,M2,N) :- abs(M1-N)<abs(M2-N).
```

Fig. 1. A simple FLUX strategy for elevator control.

The elements of hypothesized programs are called *strategy-clauses*. A strategy-clause is a non-recursive Prolog clause having an instance of $strategy(Z, A)$ as head and a body containing atoms and negated conjunctions of atoms. These atoms are defined in BK. With this definition, we assume that the relation between states and actions is functional, i.e., the action to be executed can be uniquely identified by the current state. Moreover, the absence of state update axioms and recursive definitions of strategy-clauses prohibits learned programs from looking ahead and plan.

The set of all examples E provided for the IIM contains pairs (z, a) of a state z and an action a, meaning the agent has to execute action a in state z. In this way, every example is positive. However, given the functional nature of the mapping from states to actions, an example (z, a) implicitly entails negative examples for every action other than a. This treatment of positive-only examples has already been applied in the system FILP [8].

In order to restrict the hypothesis space by expressing additional knowledge about specific domains, we use sorts, modes and occurrence restrictions, described subsequently. Moreover, we restrict the hypotheses space by a maximum *newsize* [9] to achieve finiteness.

Sorts Since the fluent calculus uses a sort signature to categorize terms, it is quite natural to use this information in a corresponding learning algorithm, too. We developed constraint handling rules to restrict variables to be of a certain sort. A relation $\succ_{sort}$ spans a tree in the set of sorts having the universal sort ANY as root. This relation enables us to compute a least general sort roughly following [10].

Modes have been used with success in a variety of ILP systems [11]. They are used to reflect the computational behavior of predicates. Arguments of

predicates can be either of input or output mode. Hypothesized clauses are required to obey mode declarations. Put simply, a variable occurrence in an input argument has to be preceded by an occurrence of the same variable in an output argument.

Occurence restrictions are used to rule out certain combinations of literals in the body of hypothesized clauses. These restrictions can be drawn from domain specific knowledge. For instance, we can express that the arguments of a predicate encoding a binary, irreflexive relation should be different.

4 Generalization

Examples, understood as pairs (z, a) of a state z and an action a, can trivially be transformed into strategy-clauses. An example $([f_1, \ldots, f_n], a_e)$ corresponds to the clause $strategy(Z, a_e) \leftarrow holds(f_1, Z), \ldots, holds(f_n, Z)$. This transformation can be understood as adding background knowledge to unit clauses, in order to compute a *relative least generalization*. The clauses serve as the input of a generalization procedure based on a sorted version of Plotkin's Least General Generalization [7], where a generalized term $lggs(t_1 : s_1, t_2 : s_2)$ is of the smallest sort s wrt. $\succ_{sort}$ such that $s \succ_{sort} s_1 \wedge s \succ_{sort} s_2$.

Since the $lggs$ of a set of clauses grows exponentially with the size of the set, we define a generalization operator gs on top of the $lggs$ producing generalizations of constant size. Literals in bodies of strategy-clauses directly or indirectly express properties of the state. This state is represented by a list. Our generalization operator is motivated by the idea that the quality of a generalization of two lists representing states depends on the order of the fluents inside the lists. This order, however, has no semantical meaning[1] and thus ordering can be seen as a task of the generalization algorithm.

Definition 1. *A generalization of two strategy-clauses using sorts:*
Let $c_1 = p_1 \leftarrow l_1 \wedge \ldots \wedge l_n$ and $c_2 = p_2 \leftarrow k_1 \wedge \ldots \wedge k_m$ be two strategy-clauses without negations such that $n \leq m$ then a generalization is given by:

$$gs(c_1, c_2) \stackrel{def}{=} lggs(p_1, p_2) \leftarrow \bigwedge W$$

where W satisfies

- $W \subseteq \{lggs(l_i, k_j) | 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge lggs(l_i, k_j)$ *is defined*$\}$
- $(\forall l_i, 1 \leq i \leq n)\, lggs(l_i, x) \in W \wedge lggs(l_i, y) \in W \supset x = y$
- $(\forall k_i, 1 \leq i \leq m)\, lggs(x, k_i) \in W \wedge lggs(y, k_i) \in W \supset x = y$

The choice of W is determined by a heuristic function $g(W, a)$, with a being the action occurring in the head of the clause $gs(c_1, c_2)$. W is chosen to maximize g.

$$g(W, a) \stackrel{def}{=} \sum_{l \in W} \Big(\sum_{k \in W \setminus \{l\}} \frac{1}{2} \sigma_{link}(l, k) \Big) + \sigma_{link}(l, a) + \frac{1}{1000} \sigma_{comp}(l)$$

[1] State composition is commutative and associative in the fluent calculus.

with

$$\sigma_{link}(x, y) \stackrel{def}{=} \text{Number of variables not of sort } STATE \text{ shared between } x \text{ and } y$$

$$\sigma_{comp}(x) \stackrel{def}{=} \text{Number of subexpressions of } x - \text{Number of variables in } x$$

In this way, we maximize the number of variable occurrences in multiple literals, while constraining the generalized clause to the size of the smallest clause involved. Variables of sort $STATE$ are ignored, since the variable Z, denoting the current state, occurs in the head of every strategy-clause and in every literal of the form $holds(F, Z)$.

Because maximizing the number of links does not necessarily lead to a unique solution, we incorporate other syntactic information into the heuristic function as well. Intuitively, from two literals which yield the same amount of links, we want to choose the more specific one. Function σ_{comp} allows comparison of literals which are not comparable using θ-subsumption, but does not contradict it. To search for the optimal W efficiently, we use A* [12]. Note that this approach is not limited to strategy-clauses, but can be applied to arbitrary horn clauses. In particular, it is designed to deal with clauses build from extensive background knowledge which usually contain many redundant literals.

The Generalization Loop

Initially, for every function symbol a into sort $ACTION$ the corresponding set of examples $\{(z, a(\boldsymbol{x}))|(z, a(\boldsymbol{x})) \in E\}$ is generalized. If a heuristic quality threshold finds the result too general, the corresponding set of examples is split into disjoint subsets. Splitting is done by either instantiating a variable in the corresponding action or by using a fluent as classificator. The first possibility yields a subset for every possible substitution of the chosen variable. The second one yields two sets of examples, one with all examples in whose state the fluent holds and one with all examples in whose state the fluent does not hold. Thus, splitting is a heuristic way of specializing the initial clauses, before the actual top-down search takes place. Generalization, evaluation and splitting are repeated until the quality threshold is reached or no further splitting is possible. The conditions under which splitting is possible also ensure that this process terminates.

5 Specialization

The specialization process searches for a correct program consisting of clauses which are each subsumed by one of the computed generalizations. To be able to introduce negations inside bodies of clauses while still maintaining top-down behavior of the search, we need a way to group multiple literals in a meaningful way. We therefore introduce *computation chains*.

Definition 2. *A computation chain is a conjunction $l_1, \ldots, l_n$ of at least one positive literal, such that for every l_i with $i < n$ at least one output argument of l_i also occurs as input argument in a literal l_j with $j > i$.*

The notion of computation chains, together with sort constraints, allows to add multiple literals to a clause at once in a meaningful manner. This reduces the effects of the plateau problem and enables the use of negated conjunctions as an expressive part of our language. The employed refinement operator $\rho\colon \mathcal{H} \mapsto \mathcal{P}(\mathcal{H})$ computes specializations by either

- unifying two variables of the same sort
- substituting a variable with a function of distinct new variables into the right sort
- adding a computation chain to the body of the clause
- adding the negation of a computation chain to the body of the clause

Note that after addition of a computation chain, the refined clause is still subject to mode restrictions. Therefore, if a variable occurs in an input argument in the chain and not in an output argument, it is bound to a variable occurring in the body of the clause. To maintain the top-down manner of the search, it is neither allowed to instantiate a variable occurring only negated nor to unify it with any other variable, when refining clauses containing negations.

The Specialization Loop

Each of the initial schemes becomes the root node of a search tree. The search trees are searched in parallel in a greedy manner, similar to the covering algorithm, which was first used in the AQ system [9].

In each search iteration, the clause with the highest heuristic evaluation in each search tree is refined and replaced by its specializations. A subsequent goal test identifies correct clauses. If a correct clause is found, it is asserted and the corresponding examples are removed.

A post-processing step is applied to both asserted clauses and the final program to remove redundant literals and clauses. A top level loop over generalization and specialization ensures completeness, if a finite hypotheses space is specified.

6 Experimental Results

We first applied the learning algorithm to the elevator control program, originally written in GOLOG [2]. Examples were generated by the strategy depicted in Figure 1. We provided predicates encoding the binary relations *unequal* and *less-than*, and the ternary relation *closer* in BK. To restrict the hypothesis space, we only used knowledge automatically derivable from a domain axiomatization.

If at least one example indicated that the elevator sometimes has to leave the first floor, a program was learned which is semantically equivalent to the one which generated the examples, otherwise learned programs always terminated once they reached the first floor. This result was stable throughout all tests.

As a more complex scenario, we chose the mailbot example, described in [13]. The main difference to the elevator scenario lies in the action related to movement. In the elevator scenario, this action has a destination as argument, while

Example Set	Solved (%)	Completeness (%)
Large Search Space		
Optimal	23	0
Good	23	0
Naive	47	25
Medium Search Space		
Optimal	24	2
Good	31	2
Naive	61	44
Small Search Space		
Optimal	18	0
Good	64	37
Naive	92	84

Fig. 2. Results in the Mailbot Scenario

the mailbot can only choose to go up or down, without initially knowing a relation between destinations and directions. Moreover, the mailbot's movement is motivated by two state properties, namely packages to be picked up and packages to be delivered.

Figure 2 shows the results of nine different experiments in the mailbot scenario. We conducted tests using different example sets and hypothesis spaces. "Optimal" denotes example sets generated by an optimal strategy which minimizes the number of actions to solve the problem. "Good" denotes example sets generated by a suboptimal, but sophisticated program, described in [13], while "naive" refers to example sets generated by a very simple strategy.

Each test was repeated using different hypothesis spaces. "Large Search Space" refers to a hypothesis space restricted only by domain dependent knowledge, i.e., sorts, modes and occurrence restrictions are direct consequences of the scenario. We provided the same predicates as we did in the elevator scenario. In the "medium search space", we removed the inequality relation and prohibited instantiations of bags and rooms in the top-down search. The "small search space" is very artificial, as we used additional restrictions not corresponding to any domain property.

Each row in Figure 2 corresponds to the evaluation of hundred learned programs. The learned programs were tested against 200 instances of the problem. A program solved the problem instance if it delivered all initial packages and terminated, otherwise the test was considered a failure. "Solved" refers to the ratio of solved problem instances by the learned programs. "Completeness" indicates the ratio of learned programs solving all test cases confronted with.

7 Summary

In this paper, we have shown a way to apply ILP techniques to learn simple Special FLUX strategies. The learning algorithm makes strong use of background

knowledge to learn programs complete and consistent with the given examples. The approach benefits from the combination of top-down and bottom-up techniques, as the top-down search does not start from unit clauses, but from a set of rather specific clauses, situated nearer potential solutions in the subsumption lattice. The techniques presented here do not depend on specific FLUX characteristics and it should be easy to adopt it to other action formalisms such as GOLOG.

This work is a first attempt to learn strategies for FLUX agents. It is strongly affected by local optima due to the rather wide search trees, i.e., the size of $\rho(C)$. This size leads to comparatively large programs incomplete wrt. the corresponding problems and is the main reason for the bad results in the mailbot scenario (see Figure 2). Reducing the influence of local optima will therefore be one of the main aspects of continuative work.

References

1. McCarthy, J.: Programs with Common Sense. In: Proc. of the Symposium on the Mechanization of Thought Processes. Volume 1., London (1958) 77–84
2. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31** (1997) 59–83
3. Thielscher, M.: FLUX: A logic programming method for reasoning agents. Theory and Practice of Logic Programming (2005) Available at: `www.fluxagent.org`.
4. Hume, D., Sammut, C.: Applying inductive logic programming in reactive environments. In Muggleton, S., ed.: Inductive Logic Programming. Academic Press (1992) 539–549
5. Moyle, S., Muggleton, S.: Learning programs in the event calculus. In: ILP '97: Proc. of the 7th Int. Workshop on Inductive Logic Programming, Springer-Verlag (1997) 205–212
6. Thielscher, M.: From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. Artificial Intelligence **111** (1999) 277–299
7. Plotkin, G.: Automatic Methods of Inductive Inference. PhD thesis, Edinburgh University (1971)
8. Bergadano, F., Gunetti, D.: An interactive system to learn functional logic programs. In Bajcsy, R., ed.: Proc. of the 13th Int. Joint Conference on Artificial Intelligence. (1993) 1044–1045
9. Nienhuys-Cheng, S.H., de Wolf, R.: Foundations of Inductive Logic Programming. Springer (1997)
10. Jr., C.D.P., Frisch, A.M.: Generalization and learnability: A study of constrained atoms (1992)
11. Lavrač, N., Džeroski, S.: Inductive Logic Programming - Techniques and Application. Ellis Horwood (1996)
12. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4** (1968) 100–107
13. Thielscher, M.: Reasoning Robots The Art and Science of Programming Robotic Agents. Kluwer Academic Publishers (2005)