When Not to Act?[⋆]

Michael Siebers

Otto-Friedrich-Universität, Professur Angewandte Informatik/Kognitive Systeme,
An der Weberei 5, 96047 Bamberg, Germany
michael.siebers@uni-bamberg.de

Abstract. In this paper we present an approach to decide which actions to avoid during automated planning. We learn avoidance expressions which hold, if the application of an action in a state will lead to a dead-end. We propose to generalize these states to clauses and then to transform them into avoidance expressions. We present a proof of concept for our approach.

Keywords: automated planning, inferring domain knowledge, dead-end states

1 Introduction

Automated planning is the "task of coming up with a sequence of actions that will achieve a goal" [6] given some problem description. As there is no general solution, this task is usually achieved by some variation of heuristic search. However, human planners may immediately "see" what they should *not* do, even if the problem itself is hard to solve for them. On the one hand, these are actions which are superfluous, like walking in a circle. On the other hand, these are actions which render solving the problem impossible, for instance, plunging into an abyss. Having such knowledge may provide valuable insight into problems. Thus, we aim at extracting such domain knowledge from problem descriptions. We will concentrate on the second kind of actions, that is actions which lead to a dead-end.

Since automated planning is formalized using first-order logic, we want to represent whether an action should be avoided as logic construct, called avoidance expression. Furthermore, we want to induce these expressions from examples using inductive logic programming.

Of course, this is not the first paper on what to avoid in automated planning. However, these approaches are either tailored to some planning system or handle only the use of such domain knowledge, not its generation. De la Rosa and McIlraith learn first-order linear temporal logic formulas to guide plan generation in TLPlan [5]. They learn conditions over state transitions, which are used to prune the search space. On a similar line Gabaldon discusses the use of control expressions to check sequence prefixes in situation-calculus-like planning [1]. Kibler and Morris handle bad decisions in the BLOCKSHEAD planner [2].

[⋆] Preliminary ideas of this paper were published by Siebers [7].

In the next section we will provide some background on automated planning and define dead-ends. In Section 3 we will introduce avoidance expressions and will show how these can be learned from problem descriptions. A first proof of concept for our approach will be given in Section 4, before we conclude in Section 5.

2 Automated Planning

In automated planning, *states* are represented as sets of positive first-order logic (FOL) literals which must be ground and function-free. Literals not in the set do not hold, as a closed world is assumed.

Actions are transitions between states. They are represented using action schemas. Every action schema $o = n(v, \varphi, \xi)$ consists of a name n, a variable list v, and two formulas, φ and ξ, which must be conjunctions of function-free FOL literals. Both formulas may only contain variables from v as free variables and must otherwise be ground. An action is an instantiation of an action schema using the variables in the variable list v. Every actions' preconditions must be consistent, that is ξ may not be a contradiction.

Action $a = (\phi, \xi)$ is said to be *applicable* in the state s iff φ holds in the state ($s \models \phi$). If $\xi = l_1 \wedge \cdots \wedge l_n \wedge \neg l_{n+1} \wedge \neg l_m$, where $l_1, \ldots, l_m$ are positive literals, then *applying* the action on state s results in $s' = (s \cup \{l_1, \ldots, l_n\}) \setminus \{l_{n+1}, \ldots, l_m\}$. For short, we write $s \xrightarrow{a} s'$ iff action a is applicable in state s and produces the *successor state* s' if applied on s.

A *planning domain* $\mathcal{D}$ is a triple (P, C, A), where P is a set of predicates with arbitrary arity, C is a set of constants, and A is a set of action schemas. A *planning problem* $\mathit{\Pi}$ is a quadruple $(\mathcal{D}, O, s_0, g)$, where $\mathcal{D}$ is the domain of the problem, O is a set of *objects*, s_0 is the *initial state* of the problem, and g is a ground, function- and quantifier-free first-order formula, called *goal*. A state is called a *goal state* if g holds in that state.

Dead-ends are states from which no solution is possible. That is, a state s is called a *dead-end* iff there exists no sequence of actions $a_1, \ldots, a_n$ and no goal state s_n such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$.

3 Avoiding Actions

Avoidance expressions describe under which conditions some previously defined action should be avoided. Since dead-ends are defined regarding a state and a goal, avoidance expressions must also be able to express relations about both. Thus, we introduce two marker predicates, :s and :g, which both may dominate single literals.

Definition 1. *The avoidance expressions for a planning domain $\mathcal{D} = (P, C, O)$ is the smallest set of expressions such that*

1. *if $p \in P$ is an n-ary predicate symbol and $o_1, \ldots, o_n$ are constants or variables, then $:\mathtt{s}(p(o_1, \ldots, o_n))$, $:\mathtt{s}(\neg p(o_1, \ldots, o_n))$, $:\mathtt{g}(p(o_1, \ldots, o_n))$, and $:\mathtt{g}(\neg p(o_1, \ldots, o_n))$ are avoidance expressions;*
2. *if ϕ and ψ are avoidance expressions, so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\exists v.\phi$, and $\forall v.\phi$.*

Additionally, avoidance expressions must be evaluated on a state and a goal. Let ϕ be an avoidance expressions. A state s and a goal g satisfy ϕ, denoted by $s, g \models \phi$, if one of the following conditions hold:

1. $\phi = :\mathtt{s}(l)$, and $s \models l$;
2. $\phi = :\mathtt{g}(l)$ and $g \models l$;
3. $\phi = \neg\psi$ and $s, g \not\models \psi$;
4. $\phi = \phi_1 \vee \phi_2$, and $s, g \models \phi_1$ or $s, g \models \phi_2$;
5. $\phi = \phi_1 \wedge \phi_2$, and $s, g \models \phi_1$ and $s, g \models \phi_2$;
6. $\phi = \exists v.\psi$, and there is some object or constant o in the planning problem of discourse such that $\psi\{v \leftarrow o\}$ holds, where $\psi\{v \leftarrow o\}$ is the result of substituting o for every free occurrence of v in ψ;
7. $\phi = \forall v.\psi$, and $\psi\{v \leftarrow o\}$ holds for every object or constant o in the planning problem of discourse.

We propose to induce avoidance expressions for an action using ILP. In the next sections we will show how examples can be transformed into a set of facts suitable for the induction of clauses, how these clauses can be transformed into an avoidance expression, and how examples can be generated from a given planning problem.

3.1 Induction

Let us, without loss of generality, assume that we induce an avoidance expressions for a specific action a. As the expression relates to a state and a goal, we will learn from state-goal pairs. For every example $e = (s, g)$, a must be applicable in s. Let E^+ denote the set of all examples, where applying a yields a dead-end and E^- the set of examples where this is not the case.

To represent examples as facts, we will employ ideas from McCarthy's situation calculus [3]. First, we introduce three sets of predicates, the state predicates P_s, the positive goal predicates P_g^+, and the negative goal predicates P_g^-, such that

1. $P_s = \{p_s(t_1, \ldots, t_n, e) \mid p(t_1, \ldots, t_n) \in P\}$,
2. $P_g^+ = \{p_{g+}(t_1, \ldots, t_n, e) \mid p(t_1, \ldots, t_n) \in P\}$,
3. $P_g^- = \{p_{g-}(t_1, \ldots, t_n, e) \mid p(t_1, \ldots, t_n) \in P\}$,
4. $P_s \cap P_g^+ = \emptyset$, $P_s \cap P_g^- = \emptyset$, and $P_g^+ \cap P_g^- = \emptyset$.

The state predicates represent the state markers and the goal predicates represent the goal markers[1]. Each of the predicates has one additional argument e,

[1] Please note that one set of predicates for the state markers is sufficient, since states are sets of positive literals.

Algorithm 1: Convert Clause to an Avoidance Expression

```
Input: a clause $p_a(e) \leftarrow \chi$
foreach $l_i$ in $\chi$ do
    if $l_i = p'(o_1, \dots, o_n)$ then
        $l'_i = \begin{cases} :\mathsf{s}(p(o_1,\dots,o_n)) & \text{if } m_s(p) = p' \\ :\mathsf{g}(p(o_1,\dots,o_n)) & \text{if } m_p^+(p) = p' \\ :\mathsf{g}(\neg p(o_1,\dots,o_n)) & \text{if } m_p^-(p) = p' \end{cases}$
    else if $l_i = \neg p'(o_1, \dots, o_n)$ then
        $l'_i = \begin{cases} \neg:\mathsf{s}(p(o_1,\dots,o_n)) & \text{if } m_s(p) = p' \\ \neg:\mathsf{g}(p(o_1,\dots,o_n)) & \text{if } m_p^+(p) = p' \\ \neg:\mathsf{g}(\neg p(o_1,\dots,o_n)) & \text{if } m_p^-(p) = p' \end{cases}$
    end
end
$\chi \leftarrow \bigvee l'_i$
foreach free variable $v$ in $\chi$ do
    $\chi \leftarrow \exists v\, \chi$;
end
return $\chi$
```

representing an unique id of the example. Furthermore, we introduce three mappings $m_s : P \to P_s$, $m_p^+ : P \to P_g^+$, and $m_p^- : P \to P_g^-$, which map the problem domains predicates to their counterparts. Then every example $e = (s, g)$ can be represented as set of facts

$$\begin{aligned} F_e = &\{p'(o_1, \dots, o_n, e) \mid m_s(p) = p' \wedge p(o_1, \dots, o_n) \in s\} \\ \cup&\{p'(o_1, \dots, o_n, e) \mid m_p^+(p) = p' \wedge g \models p(o_1, \dots, o_n)\} \\ \cup&\{p'(o_1, \dots, o_n, e) \mid m_p^-(p) = p' \wedge g \models \neg p(o_1, \dots, o_n)\} . \end{aligned}$$

Additionally, we introduce an unary predicate p_a, which denotes, that the action a should be avoided in example e. Positive examples for the induction problem are $\{p_a(e) \mid e \in E^+\}$ and negative examples are $\{p_a(e) \mid e \in E^-\}$. Background knowledge is the union of the fact representations of all examples. Using this information, a set of clauses for the avoidance of action a can be induced with an general purpose ILP system.

A learned set of clauses can be transformed to an avoidance expression. Every clause has the form $p_a(e) \leftarrow \chi$, where $\chi = l_1 \wedge \dots \wedge l_n$. As first step the example id is removed from each literal l_i. Then, goal predicates are transformed to goal markers and states predicates are transformed to state markers. Finally, the formula is existentially quantified for every free variable occurring in χ. The avoidance expression for action a is the disjunction of all resulting formulas. Details on the conversion can be found in Algorithm 1.

Algorithm 2: Collect Examples

```
Input: The initial state $s_0$, the set of actions $A$
initialize $Q$ to a queue holding only $s_0$
while $Q$ is not empty do
    $s$ = pop from $Q$
    if $s$ is no goal state then
        foreach action $a \in A$ that is applicable in $s$ do
            $s \xrightarrow{a} s'$
            if $s'$ is a dead-end then
                store $(s, a)$ as disallowed example
            else
                store $(s, a)$ as allowed example
                push $s'$ on $Q$
            end
        end
    end
end
```

3.2 Example Generation

Examples for learning avoidance expressions can be generated from simple problems within the problem domain. For such problems, a exhaustive exploration of the state-space is possible. Starting from the initial state s_0 a basic breadth-first search with loop detection is performed. For each state, all possible successor states are constructed. Then, each successor state is tested whether it is a dead-end or not using depth-first search with loop detection. If it is a dead-end, the former state and the action that lead to the successor state are stored as disallowed pair. If the successor state is no dead-end the pair is stored as allowed pair and the successor state is added to the search frontier. A goal state is not expanded further, but simply removed from the search frontier. The algorithm is shown in Algorithm 2. The examples for some action a are inferred from the stored pairs, such that $E^+ = \{(s, g) \mid (s, a) \text{ is disallowed}\}$, and $E^- = \{(s, g) \mid (s, a) \text{ is allowed}\}$.

4 Proof of Concept

We implemented the avoidance learning approach and applied the implementation on variants of the rocket domain introduced by Veloso [8]. The rocket domain deals with a number of boxes positioned on earth and a rocket which can fly to the moon. The aim is to bring all or certain boxes to the moon.

4.1 Domain and Problems

We modeled the domain with three unary predicates (`on-earth`, `on-moon`, and `loaded`). Furthermore, there are 5 action schemas:

$$\exists x.{:}\texttt{s}(\texttt{on-earth}(x)) \wedge {:}\texttt{g}(\texttt{on-moon}(x)))\vee$$
$$\exists x.{:}\texttt{s}(\texttt{loaded}(x)) \wedge {:}\texttt{g}(\texttt{on-earth}(x)))$$

Fig. 1. Learned avoidance expression for `fly` in the rocket domain

load_earth(box) Load a box into the rocket if both are on earth,
load_moon(box) Load a box into the rocket if both are on moon,
load_earth(box) Place a previously loaded box on earth,
load_moon(box) Place a previously loaded box on moon,
fly Fly the rocket and its contents to the moon.

We defined a total of 25 problems with 1 to 10 boxes. In the *basic problems* all boxes have to be on the moon in the goal state. In the *excluding problems* some specific boxes must be on the moon, while the others have to remain on earth. Finally, in the *any problems* some unspecified boxes had to be on the moon. In every problem all boxes and the rocket were on earth initially.

4.2 Avoidance Learning

To test our avoidance expression learning system we induced an avoidance relation on the rocket domain for all actions schemas using FOIL [4]. We learned from three small basic problems and one small excluding problem. The first three had 1,2, or 3 boxes, respectively. The last problem had 3 boxes, where two had to be on the moon while one had to remain on earth.

From these problems 122 examples were created and one avoidance expression was learned. The learning process took 12 ms. The learned avoidance expression for action fly can be seen in Figure 1. Informally it states "don't fly if anything is on earth which must finally be on the moon" and "don't fly some box to the moon if it has to stay on earth". All other actions are reversible and may thus cause no dead-end. As expected, no avoidance expressions were induced for these actions.

5 Conclusion and Next Steps

We defined syntax and semantics for action avoidance expressions. Additionally, we showed an algorithm for learning avoidance expressions from given domain and problem specifications. We evaluated both within the rocket domain. The learning component performed extremely well. Leading to a good solution in very short time.

Further work should concentrate on evaluating the system. Evaluation should be done within different domains with varying number of problems. First improvements could consider the dead-end detection. This is currently only a naive depth search. More sophisticated approaches (like using some planning system) are possible.

References

1. Gabaldon, A.: Precondition control and the progression algorithm. In: Proceedings of the 9th International Conference on Knowledge Representation and Reasoning. pp. 634–643. AAAI Press (2004)
2. Kibler, D.F., Morris, P.: Don't be stupid. In: Hayes, P.J. (ed.) IJCAI. vol. 1, pp. 345–347. William Kaufmann (1981), `http://ijcai.org/Proceedings/81-1/Papers/064.pdf`
3. McCarthy, J.: Situations, actions, and causal laws. Tech. Rep. Memo 2, Stanford Artificial Intelligence Project, Stanford University (1963), classic
4. Quinlan, J.: Learning logical definitions from relations. Machine Learning 5(3), 239–266 (1990)
5. De la Rosa, T., McIlraith, S.A.: Learning domain control knowledge for TLPlan and beyond. In: Proceedings of the ICAPS-11 Workshop on Planning and Learning (2011)
6. Russel, S.J., Norvig, P.: Artificial Intelligence. A Modern Approach. Pearson Education, second edn. (2003)
7. Siebers, M.: Transfer of domain knowledge in plan generation: Learning goal-dependent annulling conditions for actions. KI 28(1), 35–38 (2014)
8. Veloso, M.M.: Nonlinear problem solving using intelligent casual-commitment. Tech. rep., Carnegie Mellon University (December 1989)