Learning Opposite concept for Machine Planning

Kang Soo Tae

Department of Computer Engineering

Jeonju Universit

1200 3-Ka Hyojadong Chonju, 560-759 Korea

kstae@www.jeonju.ac.kr

Abstract [1]

An incomplete planning domain theory can cause an inconsistency problem in a noisy domain. To solve the problem of applying two opposite operators to the same state, we present a novel method to learn a negative precondition as control knowledge. Even though the control knowledge is unknown to a machine, it is implicitly known as opposite concept to a human. To learn the human concept, we propose a new technique to mechanically generate a graph composed of opposite operators from a domain theory and extract opposite literals. We show that the opposite concept is a special type of mutex used in Graphplan. A learned concept can simplify the operator by removing a redundant precondition while preventing inconsistencies.

Introduction

A domain theory constitutes a basic building block for a planning system. However one of the hard problems in planning is that a machine does not know the interpretation of sentences in the theory. For example, Graphplan uses mutual exclusion relations called mutex, but currently it cannot infer *not* and just treats *not* as a string of characters. If we have an operator requiring P to be false, then we need to define a new proposition Q that happens to be equivalent to (not P). For instance, if P is (on-ground <y>), then w might have Q be (not on-ground <y>), or (not-on-ground <y>), or (up-in-the-air <y>) (Brum and Furst, 1997). Understanding intelligent entities is a hard but fundamentally important AI problem. One approach fo building an intelligent system is to study a human as an example.

In this paper, we will investigate a subtle aspect of an incomplete domain theory that is related with a domain expert's implicit knowledge. First, we introduce an inconsistency problem in which incomplete ly-specified opposite operators are applied incoherently. Then, by adopting a three-valued logic in learning preconditions, we show how to learn a negative precondition that ca detect an inconsistency (Tae and Cook 1996). As a next step, observing that a human possesses certain knowledge that detects an inconsistency easily and almost unconsciously, we propose an approach that automatically extracts this kind of human knowledge from a graphical domain theory and applies the new concept in order to make an operator definition more compact.

Learning a Negative Precondition

A planning domain theory specifies an agent's legal actions in terms of a set of operators. A Strips-lik operator models an agent's action in terms of a set of preconditions *pre(op)*, an add-list, *add(op)*, and a delete-list, *del(op)*. In order to apply an operator, the operator's positive and negative preconditions must be satisfied in the internal state of the agent. We will first introduce a previous method that learns an operator's preconditions directly from a two-valued state and point out the resulting inconsistent planning problem, where two opposite operators can be applied to the same state in a noisy domain. To solve this problem, we introduce a ne method of learning a negative precondition from a three-valued state. A learne negative precondition detects an inconsistent state and functions as control knowledge for not executing an operator.

Inconsistency Problem of a Machine

A state is conventionally described by a two-valued logic, where a fact is either true or false in the state. Based on the Closed-World Assumption (CWA), if p is false, $\sim p$ is assumed to hold. Using CWA, OBSERVER learns an operator through observing how states change while an expert solves a problem (Wang 1995). If an operator is successfully executed in a state satisfying all the necessary positive and negative preconditions of the operator, the state constitutes a positive training example for learning the operator. OBSERVER learns initial preconditions b parameterizing each predicate in the state according to type information. Learned from a single training example, the initial preconditions may contain irrelevant literals. OBSERVER generalizes this overly-specific preconditions by removing irrelevant literals through observing more training examples. If a predicate does not appear in a new training example, the predicate is removed from the real preconditions.

However, learning an operator simply from a state using CWA, OBSERVER is unable to learn a negative precondition. OBSERVER's incomplete operator faces

some inconsistency problems in a noisy domain. Suppose that an agent's arm is empty in the actual world. If the agent uses noisy sensors, the agent may internally believe that its arm is empty and that it is also holding an object at the same time: {*arm-empty*, (*holding x)*}. Strangely, a machine may not detect that it is an impossible state. For example, if a state, {(*holding box1), (arm-empty), (next-to robot box1), (carriable box1*)}, is supplied, PRODIGY (Carbonell et al. 1992) cannot detect that it is inconsistent and it may try to execute a wrong operator. Let the preconditions of **picku** be {(*arm-empty), (next-to robot box), (carriable box*)} and those of **putdown** be {(*holding box*). If the goal is {(~ *arm-empty*)}, PRODIGY generates a plan, *(pickup box1)*, while if the goal is {(~ *holding box1*)}, it generates another plan, *(putdown box1)* from the initial state. This shows that if a planner is equipped with incomplete domain knowledge, planning is unreliable in a complex domain. On the other hand, note that if *arm-empty* is true in a state, a human can infer that ~ (*holding x*) also holds at the same time. Thus, he can easily perceive that the above belief, {*arm-empty*, (*holding x)*}, is inconsistent containing opposite literals {(*holding x), ~(holding x*)}. Note that a negative precondition, used as crucial control knowledge in a machine, is rather obvious and redundant to a human, and we will focus on this matter in the next section.

Negative Precondition as Control Knowledge

To solve this kind of inconsistency problem, we present a method of learning a negative precondition and using the learned precondition as a machine's control knowledge. WISER (Tae and Cook 1996) is an operator learning system running on top of PRODIGY and actually learns a negative precondition. Note that while a state is described by a two-valued logic, an operator is described by a three-valued logic. If *p* is not in the preconditions of an operator, *p* in a state is irrelevant in applying the operator. If *p* should not hold in a state, ~*p* must explicitly appea as an operator's precondition. To learn such a negative precondition from a state, we need t describe a state by a three-valued logic. For that purpose, we first transform the state into its closure by releasing CWA.

Let *PRED** represent the space of all the predicates known to WISER. Let *S* be a positive state, *P* be the set of predicates true in *S*, and *N* the set of predicates not true in *S*. Since *{PRED* - P}* represents the set of predicates which are not true by CWA, it corresponds to *N*. Releasing CWA, *S* transits to its closure, $S^* = P + Neg(\{PRED^* - P\})$, where *Neg(X)* means the negated value of *X*. *S* is identical to *S*, but it provides a more comprehensive description of the same state by comprising negative literals. *S** constitutes an overly-specific definition for inducing preconditions in WISER. WISER generalizes overly-specific preconditions by eliminating irrelevant literals through experimentation by adopting a bottom-up search generalization method (Craven and Shavlik 1994). While the preconditions are overly-specific, WISER negates each literal in the initial definitions, $l \in S^*$, transitioning to a new state $S_{new} = \{S^* - l + \sim l\}$. If the operator is still applicable, WISER deletes the irrelevant literal, *l*, from the overly-specific preconditions. Note that while an irrelevant literal is assigned the symbol * in a state description *a priori* in Oates and Cohen's work, WISER can detect a n irrelevant literal through experimentation.

To illustrate this method through an example, let *PRED*={A, B, C, D, E, F}*, and the real preconditions *Pre(op)* of an operator *op* be *{A, B, C, ~D}*. Let a positive state S_0 be *{A, B, C, E}*. OBSERVER initializes the preconditions as *{A, B, C, E}*. Given another positive state *{A, B, C, F}*, OBSERVER deletes *E* and generalizes the preconditions to *{A, B, C}*. On the other hand, WISER initializes the preconditions to *{A, B, C, ~D, E, ~F}*. To generalize the overly-specific initial preconditions through experiments, WISER negates each literal in the initial definitions one at a time. If the operator is still applicable, WISER deletes the literal from the preconditions. Given a new state, S_1 = *{ ~A, B, C, ~D, E, ~F}*, since *op* cannot be applied to S_1, WISER learns that *A* is relevant. Similarly, *op* cannot be applied to another state S_2 = *{A, B, C, D, E, ~F}*, and WISER learns that *~D* is also relevant. When *op* is successfully applied to S_3 = *{A, B, C, ~D, ~E, ~F}*, WISER learns that *E* is not relevant and deletes the literal from the preconditions. In this way, WISER generalizes th initial precondition, *{A, B, C, ~D, E, ~F}*, to *{A, B, C, ~D}*. Given a state, *{A, B, C, D}*, *Pre(op)* is not met, and the operator must not be fired. Note that while *op* internall fires in OBSERVER, it is not fired in WISER

Finally, let's show how WISER can solve the previous inconsistency problem after learning a negative precondition. Suppose the inconsistent state is supplied to an agent by noisy sensors: {(*holding box1), (arm-empty), (next-to robot box1), (carriable box1*)}. Given the expert-generated preconditions of **picku** and **putdown**, if the goal is {~*arm-empty*}, PRODIGY generates a plan, *(pickup box1)*, and if the goal is {~(*holding box1*)}, it generates another plan, *(putdown box1).* The plan execution sometimes succeeds and sometimes fails due to a perceptual alias (Benson 1995). Using the above algorithm, WISER successfully generates more constrained preconditions of **picku** : *{(~holding box), (arm-empty), (next-to robot box), (carriable box)}.*

Learning Opposite Concepts

In the previous section, we observed that a human possesses knowledge that is unknown to a machine and he/she can immediately detect an inconsistent state. To mechine-learn this type of knowledge, we suggest a method to generate opposite operators from a graph in a domain theory and extract opposite propositions through experimenting the operators. The learned concept simplifies an operator b removing redundant negative preconditions while preventing inconsistency We show that this opposite

concept is a mutex in Graphplan.

Machine and Implicit Human Knowledge

Using a negative precondition raises a question of wh explicitl encoding control knowledge is necessary to a machine while it i unnecessary to a human. Suppose a state description S_1 includes two predicates *p* and *q*. If a rule *R: p* $\rightarrow$ *q* is known for system *A*, another state description S_2 is obtained by removing *q* from S_1. S_1 and S_2 are equivalent with respect to the rule. On the other hand, suppose the rule is not known to another system *B*. Since *B* cannot infer *q* from *p*, S_2 is not equivalent to S_1 and not encoding *q* in S_2 may cause a problem. For instance, suppose a simple rule *(dr-open dr)* $\rightarrow$ $\sim$*(dr-closed dr)* is known to a human. Then, S_1 = {*(dr-open dr), ~(dr-closed dr), (next-to robot dr)*} and S_2 = {*(dr-open dr), (next-to robot dr*)} are equivalent, and $\sim$ *(dr-closed dr) in* S_1 is redundant. On the other hand, if the rule is not known to a planning system, the negative literal is not known to the system in S_2.

Knowledge acquisition is mapping of expert knowledge to a machine. However, after mapping, the expert may possess some knowledge not captured in a planning system (desJardins 1992). If an expert wrongly assumes that a planning system knows the rule and S_1 and S_2 are equivalent states to the system, the domain theory that he/she build may cause an inconsistency problem as shown previously A type of incompleteness in a domain theory may occur due to certain types of expert knowledge which a machine does not possess after knowledge mapping, but which the expert assumes that th machine possesses. This type of exper knowledge is called *implicit* knowledge. Since we are not yet at the level of scientifically understanding how the human mind works, especiall at the level of unconsciousness, it is difficult to analyze the complicated structure of an expert's implicit knowledge and make it explicit for a machine. But, as a first step, we will focus on a somewhat simple problem of understanding an opposite concept. Note that an opposite concept can be used to infer a negative fact from a positive fact. For example, if a door is open, it can be inferred that the door is *not* closed. An expert can initially encode an opposite concept into the domain theory as an inference rule (Minton 1988) or as an axiom (Knoblock 1994). However, it is overwhelming to manuall encode all the related opposite concepts in a complex domain. Thus, an adaptive intelligent agent should be able to lear an opposite concept autonomousl in a new situation.

Suppose that a domain expert does not encode opposite concepts into the domain theory as shown in PRODIGY. Then, while the expert unconsciously uses an opposite concept, a system cannot infer a negative literal. Fo example, if a door is open, the expert understands that the door is not closed, and if a state includes both *door-open* and *door-closed*, he knows that the state is inconsistent . But a current symbolic planning system like PRODIGY, which does not understand opposite concepts, cannot detect an inconsistent state. While PRODIGY's simple theory operates in a noiseless domain, this causes a problem in a complex domain. Building a system with an erroneous assumption that the system understands human concept s can cause unexpected serious problems.

Finding Opposite Operators

An operator corresponds to an action routine of a robot (Fikes, Hart, and Nilsson 1972). Since each routine can be processed independently from other routines, each operato is also an independent module in the domain theory. However, even though the operators are unrelated to each other on the surface, they can be closely related in a deep structure of human percept. For example, the *open-dr* and *close-dr* operators are conceptually seen as opposite. We suggest a technique to find opposite relation s existing between special type of operators and to simplify them syntactically by removing redundant negative preconditions

The set of operators in a domain theory can be divided into two congruent groups based on an operator's effects on its target object: *temporary* and *destructive* operator groups. When an operator is applied to a target object, the state *S* of the object may change. If the operator's effect on the object is not permanent, then the operator is classified as *temporary*. Applying a series of other operators can restore *S*. Thus, the same operator can be applied to the same object again. On the other hand, if an operator's effect on the target object is permanent and the original state cannot be restored, the operator is classified as destructive. Note that i f a temporary operator is to be repeatedly applied to the same object, some other temporary operators must restore the operator's preconditions satisfied at the original state. In fact, the other operators undo the effect of the operator on the object. If they do not exist in the domain, the effects of the operator on the target object may remain permanent and this domain is useless.

For example, let a domain theory be composed of two temporary operators, *open-dr* and *close-dr* and a destructive operator, *drill*. When *open-dr* is applied to open a closed door, the original state of the door can be restored b applying *close-dr*. Thus, *open-dr* can be applied again to the door. Note that *close-dr* restores the preconditions of *open-dr* by undoing the effects of *open-dr*. On the other hand, for a destructive operator, *drill*, the change to the state on the target object is designed to be permanent, and other operators must not undo the effects.

To investigate some interesting relationship between two temporary operators, P and Q, such that P undoes the effects of Q on a target object as well as it restores the preconditions of Q, we generate a dependency graph between the effects of an operator and the preconditions of another operator. For an operator, *op*, let *prestate(op)* be a state which satisfies *pre(op)*, the preconditions of *op*, and let *poststate(op)* be the state occurring after applying *op* at *prestate(op)*. *poststate(op)* is calculated by the operation: *prestate(op) + add(op) - del(op)* . The domain theory is structurally represented as a directed graph, *D = (V, E)*, where $V = \{op_1, ..., op_m\}$ and $E=\{e_1, ..., e_n\}$. An edge $e_{ij} \in E$ connects one operator op_i to another operator op_j if

poststate(op_i) satisfies *pre(op_j)*. e_{ij} indicates that op_j can be always applied immediately afte op_i was applied.
Let's consider a set of operators: *open-dr, close-dr, lock-dr,* and *unlock-dr*. There is an arc from *open-dr* to *close-dr* because applying close-dr always satisfies the precondition of open-dr, and we can always open the door immediately after *close-dr* is applied. Since there is an arc from *open-dr* to *close-dr* as well, there is a cycle composed of *close-dr* and *open-dr*. Similarly, there is a cycle composed of *lock-dr* and *unlock-dr*. However, there is no arc from *close-dr* to *lock-dr* because if a robot does not hold a key yet, it needs to subgoal to *pick-up* a key before it locks the door.
For an *n-cycle*, a cycle composed of *n* operators, an arc $e_{i,\ (i+1) \bmod n}$, for $i = 1, ..., n$, connects op_i to $op_{(i+1) \bmod n}$. The arc represents that *poststate(op_i)* satisfies the preconditions of $op_{(i+1) \bmod n}$. Thus, *poststate(op_i)* obviously becomes *prestate($op_{(i+1) \bmod n}$)*.

Theorem: A temporary operator belongs to an *n-cycle*.

Proof) Let *op* be a temporary operator. Given *prestate(op), poststate(op)* is obtained by applying *op* to *prestate(op)*. If *pre(op)* still holds after applying *op*, then *pre(op)* $\subset$ *poststate(op)*. *Thus, poststate(op)* becomes *prestate(op)* and there is an arc from *op* to itself as a vacuous self-loop. On the other hand, if *pre(op)* does not hold after applying *op*, then *pre(op)* *poststate(op)*. Let $P = \{p_1, ..., p_k\}$ be the literals that existed in *prestate(op)* but which disappear in *poststate(op)* after applying *op*. To apply *op*, a temporary operator, again to the object, *prestate(op)* must be restored. Hence, there exists a sequence of operators $op_j, ..., op_n$ that establishes *P*, where op_j immediately follows *op*. Thus, there is a path from *op* to op_n. Since *op* can be applied immediately after the sequence of operators are applied, *poststate(op_n)* must satisf *prestate(op)*, and there is an arc fromop_n to *op* □

As a special case of an *n*-cycle, a 2-cycle, composed of two operators, forms a bipartite complete graph. For any two operators forming a cycle, let *Dual* for an operator be the function that returns the other operator in the pair. If op_i and op_j form a cycle, *Dual(op_i)* is op_j and *Dual(op_j)* is op_i. *Dual(op)* establishes th preconditions that *op* has deleted. Restoring the preconditions is done by undoing the effects of *op*, that is, by deleting what were added b *add(op)* and adding again what were deleted b *del(op)*. Recursively, *Dual(Dual(op))*, which is actually *op*, restores the preconditions of *Dual(op)* by undoing the effects of *Dual(op)*. Note that *prestate(op)* is the same as *poststate(Dual(op))*, and *prestate(Dual(op))* is the same as *poststate(op)*. We ca easily show that the add list of one operator is the same as the delete list of its dual operator. From the formula, *poststate(op) = prestate(op) + add(op) - del(op)*, we deduce *prestate(op) = poststate(op) - add(op) + del(op)*, which is the same as *prestate(Dual(op)) - add(op) + del(op)*. Note that *'- add(op)'* functions as the delete list of *Dual(op)* while *'+ del(op)'* functions as the add list of *Dual(op)*. Thus, we showed that *add(op)* $\equiv$ *del(Dual(op))* and *del(op)* $\equiv$ *add*(*Dual(op)*).

What does it mean that *Dual(op)* adds what *op* deleted and deletes what *op* added? Since the adding and deleting of a literal to a state is the opposite operation, *op* and *Dual(op)* constitute the *opposite* function. Two operators are defined as opposite operators *iff* the add list of one operator is the same as the delete list of the other operator and the delete list of one operator is the same as the add list of the other operator. The opposite operators undo the effects of each other. For example, *add(Open-dr)* is {*door-open*} and *del(Open-dr)* is {*door-closed*}, while *add(Close-dr)* is *{door-closed}* and *del(Close-dr)* is *{door-open}*. Thus, *Open-dr* and *Close-dr* constitute the opposite operators.
Note that two opposite operators are closely related to a binary mutual exclusion relation (mutex) used in Graphplan. Two actions in Graphplan (operators in our discussion) at the same level are mutex if either 1) the effect of one action is the negation of another action's effect or 2) one action deletes the precondition of another, or 3) the actions have preconditions that are mutually exclusive. We conjecture that if any two operators satisfy all three conditions, the form opposite operators.

Using Opposite Literal for Simplificatio

We will show how to extract opposite literals from opposite operators using an experimentation method and use them to remove redundant negative preconditions.
Let op_i and op_j be opposite operators. *add(op_i)* is opposite to *add(op_j)*, and *add(op_i)* = $\{p_1, ..., p_n\}$ contains a literal whic is opposite to another literal in *add(op_j)* = $\{q_1, ...,q_m\}$. If a literal $p_i \subset$ *add(op_i)* is the opposite concept to a literal $q_k \subset$ *add(op_j)*, a state $\{p_i, \sim q_k\}$ is feasible, but $\{p_i, q_k\}$ is inconsistent and it is not feasible as a state. To find the opposite literals through experimentation, an initial state *S* is set as $\{p_i\}$ in $\{p_1, ..., p_n\}$ one at a time, for each $i = 1, ..., n$, and then we insert into *S* each literal q_k from $\{q_1, ...,q_m\}$ one at a time, for $k = 1, ..., m$. When attempting to insert q_k to *S*, if $\{p_i, q_k\}$ is not possible and causes the state to change p_i to $\sim p_i$, resulting a unexpected state $\{q_k, \sim p_i\}$, then q_k and p_i are the opposite literals, and $\sim p_i$ can be inferred from q_k, thus creating a rule $q_k \rightarrow \sim p_i$. For example, suppose *lock-dr* and *unlock-dr* are the opposite operators. Let *add(lock-dr)* be *{locked}*, and *add(unlock-dr)* be *{unlocked}*. If *S* = *{locked}* is the initial state, adding *unlock* to *S*, *{locked, unlocked}* is not possible and the state changes to a new state *{unlocked, ~locked}*, thus a rule *unlocked* $\rightarrow \sim$ *locke* is learned by experiments.
In Graphplan (Brum and Furst, 1997), two propositions ar mutex if one is negation of the other, or if achieving the preconditions are pair-wise mutex. Note that if any tw propositions satisfy both of the conditions, they form opposite literals. Learning the opposite concept as a rule simplifies an operator definition because a negative literal can be inferred from a positive literal. For the noise-proof preconditions of *pickup*, *{(arm-empty), ~(holding x), (next-to robot box), (carriable box)}*, if a rule *(arm-empty)* $\rightarrow$ *~(holding x)* is learned, WISER can generate more simplified preconditions of *pickup*: *{(arm-empty), (next-to*

robot box), (carriable box)}. When applied as a preprocesso to an incomplete domain theory, this approach of learning rules simplifies the domain theory as well a makes the theory more complete.

Further Research and Conclusion

A planning domain theory represents an agent's knowledge about the task domain. We presented a method to learn a negative precondition to detect a problem in which inconsistent operators can be applied to the same state. Next, from observing that a human can immediately detects an inconsistent state, we investigate a type of implicit human knowledge, called opposite concept. First, we generate a graph composed of two operator where one operator deletes the other operator's preconditions and effects, and then we show how to extract opposite propositions through experimentation. The opposite operators and propositions are a special type of mutex used in Graphplan's algorithm. While mutex is a procedural inference and current systems cannot understand the concept of *not*, we conjecture that understanding an opposite concept is fundamental for an agent to survive in the real world. We will implement and test opposite concept as the next step and our human-oriented system will become more intelligent.

References

Benson, S. Inductive Learning of Reactive Action Models, in *Proceedings of the 12th International Conference on Machine Learning,* 1995.

Brum, A. L. and Furst, M L., Fast Planning through Planning Graph Analysis, in Artificial Intelligence 90(1-2): 281-300, 1997.

Carbonell, J. G., Blythe, J., Etzioni, O., Gil, Y., Knoblock, C., Minton, S., Perez, A., and Wang, X. PRODIGY 4.0: The Manual and Tutorial. *Technical Report CMU-CS-92-150*, Carnegie Mellon University, Pittsburgh, PA, 1992.

Craven, M. W. and Shavlik, J. W. Using Sampling and Queries to Extract Rules from Trained Neural Networks, in *Proceedings of the 11th International Conference on Machine Learning,* 1994.

DesJardin , M. Knowledge Development Methods for Planning Systems, in *AAAI-94 Fall Symposium Series: Planning and Learning: On to Real Applications,* 1994.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. Learning and Executing Generalized Robot Plans, in *Artificial Intelligence 3,* 1972.

Gil, Y. 1992. Acquiring Domain Knowledge for Planning by Experimentation. Ph.D. Dissertation., Carnegie Mellon Univ.

Knoblock, C. A. Automatically Generating Abstractions fo Planning, in *Artificial Intelligence,* 68, 1994.

Minton, S. Learning Search Control Knowledge: An Explanation-Based Approach, Kluwer Academic Publishers, Boston, MA, 1988.

Ourston D. and Mooney, R. J., Theory Refinement Combining Analytical and Empirical Methods, in *Artificial Intelligence,* 66, 1994.

Pearson, D. J. Learning Procedural Planning Knowledge in Complex Environments, Ph. D. Dissertation, University of Michigan, Ann Arbor, MI, 1996.

Smith, D. and Weld, D., Conformant Graphplan, in *Proceedings of 15th Nat. Conf. AI*, 1998.

Tae, K. S., and Cook, D. J. Experimental Knowledge Acquisition for Planning, in *Proceedings of the 13th International Conference on Machine Learning,* 1996.

Tae, K. S., Cook, D. J., and Holder, L. B Experimentation-Driven Knowledge Acquisition fo Planning, to appear in *Computational Intelligence* 15(3), 1999.

Wang, X. 1995 Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition, in *Proceedings of the 12th International Conference on Machine Learning,* 1995.

Weld, D. Recent Advances in AI Planning, to appear in *AI Magazine,* 1999.