에이전트의 부정에 대한 개념 학습
(Agent's Learning Concept for Negation)

태 강 수 †
(Kang Soo Tae)

요 약 영역이론의 숨겨진 문제점들 중의 하나는 에이전트가 자신의 행위를 이해하지 못한다는 점이다. Graphplan은 효율향상을 위해 mutex를 활용하고 있지만 이와 관련된 부정의 의미를 이해하지 못함으로써 영역이론의 중복성 문제를 야기한다. 이 문제에 대한 해결을 위해 IPP에서는 not 등과 같은 부정함수를 이용하지만, 부정함수의 사용은 시간과 공간적 비용을 수반한다. 인간은 주어진 어떤 사실을 부정하기 위하여 MDL 원리에 의해 반대개념을 사용한다는 점을 통하여, 우리는 부정적 사실을 표현하기 위해서 통념적 방식처럼 부정함수를 사용하는 것보다 긍정적 atom을 사용하는 것이 지능에이전트의 구축을 위해서 더 효율적 기법이라는 가설을 제시하고 IPP 도메인에서 이 가설을 지지하는 실험적 결과를 제시한다. 인간이 사용하는 것과 유사한 반대개념을 에이전트가 자동적으로 학습하기 위하여 영역이론으로부터 반대 연산자들로 구성된 사이클을 생성하고 연산자들에 대한 실험을 통해서 반대 literal들을 추출한다.

Abstract *One of the hidden problems in a domain theory is that an agent does not understand* the meaning of its action. Graphplan uses mutex to improve efficiency, but it does not understand negation and suffers from a redundancy problem. Introducing a negative function not in IPP partially helps to solve this kind of problem. However, using a negative function comes with its own price in terms of time and space. Observing that a human utilizes opposite concept to negate a fact based on MDL principle, we hypothesize that using a positive atom rather than a negative function to represent a negative fact is a more efficient technique for building an intelligent agent. We show empirical results supporting our hypothesis in IPP domains. To autonomously learn the human-like concept, we generate a cycle composed of opposite operators from a domain theory and extract opposite literals through experimenting with the operators.

1. Introduction: Motivation of research

A domain theory constitutes a basic building block for a planning agent. However, one of the hard problems in a domain theory is that an agent does not understand the meaning of its action. A planner like Graphplan [1] uses mutual exclusion relations, called mutex, to infer inconsistency between two operators or predicates, but the planner does not understand negation, which is closely related with mutex, and just treats a negative term as a string of characters. If an operator requires P to be false, then Graphplan defines a new proposition, say *not-P* or Q, which happens to be equivalent to (*not* P). This kind of seemingly simplistic negative notation may cause a redundancy problem such that a state change is represented by two processes, such as add(not-in(x)) and del(in(x)), unless an agent is equipped with a special inference knowledge. Unfortunately, most current systems do not possess this level of knowledge yet.

Recent very efficient planning systems like IPP [2], standing for Interference Progression Planner, or Blackbox [3] introduce a negative term, like *not* or !, to extend Graphplan to handle a negative function. Thus, (not on-ground <y>) can be used instead of

· 본 연구는 전주대학교 학술연구비 지원에 의해 수행되었음.
† 정 회 원 : 전주대학교 컴퓨터공학과 교수
kstae@jeonju.ac.kr
논문접수 : 1999년 7월 20일
심사완료 : 2000년 3월 25일

(not-on-ground <y>) to negate a fact in a domain. A negative term enables an agent to represent an operator's effect by a single process such as add(not in(x)). Note that using *not* to negate a predicate is a syntactic level of knowledge that simply adds a negative symbol in front of a predicate. Thus, while the agent can easily distinguish syntactic opposite terms like *on* and *not on*, it cannot distinguish semantic opposite terms like *on* and *off*.

We believe that it is fundamentally important to build an intelligent agent that is able to reason this kind of opposite relation between two positive terms. In this paper, we first point out that a human being tends to represent a negative fact by a positive predicate rather than using a negative function as in IPP. For example, we use *wet* more frequently than *not dry* to negate *dry* as a short cut. We usually call this type of positive representation as using an opposite concept, and the original Strips representation as well as Graphplan adopts this approach. Based on a human model, we will argue that using Strips-like *not-P* notation can be a more efficient and powerful technique for building an intelligent agent than using the (*not P*) notation. We demonstrate some empirical results in IPP domains that support our hypothesis in terms of time and space. Note, however, that the critical difference between a human and a Strips-like planner is that only the former possesses implicit knowledge that can infer that *P* and *not-P* are opposite. For the purpose of building an intelligent agent, we propose a method to machine-learn this human-like opposite concept, such as Opposite(*wet*)→*dry*, from a graphical domain theory.

2. Issue of Representing Negative Fact

We first introduce two current approaches of representing a negative fact as an operator's effect and show the problems with each approach. Then, we introduce human-like opposite concept as a solution based on the principle of Minimum Description Length [4].

2.1 Two negative representations

A Strips-like operator models an agent's action in terms of preconditions, *pre(op)*, and effects which are in turn composed of an add-list, *add(op)*, and a delete-list, *del(op)* [5]. To apply an operator, all of its positive and negative preconditions must be satisfied in the internal state of the agent. An operator's negative effects can be represented either as a del (P) or add (not P).

Two operators (actions) in Graphplan at the same level are mutex if either 1) the effect of one action is the negation of another action's effect, 2) one action deletes the precondition of another, or 3) the actions have preconditions that are mutually exclusive. However, Graphplan 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 *not-P* that happens to be equivalent to (*not P*). For instance, if *P* is *(on-ground <y>)*, then we might have *not-P* be *(not on-ground <y>)*, or *(not-on-ground <y>)*, or *(up-in-the-air <y>)* [1].

Suppose that an agent's arm is empty in the actual world. If the agent's sensors are noisy, 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)*}. A machine with incomplete domain knowledge cannot detect that it is an impossible state. Note, however, that if *arm-empty* is true in a state, a normal human can infer that ~(*holding x*) also holds at the same time. Thus, he / she can easily perceive that the above belief is inconsistent containing opposite literals: {*arm-empty*, (*holding x*), ~(*holding x*)}. Even though the process of inferring a negative predicate from a positive predicate seems rather self-obvious to a human, it can be used as crucial control knowledge in a machine. We will focus on this matter in a later section.

If a predicate *(on-ground <y>)* should be deleted when applying an operator, both *Del(on-ground <y>)* and *Add(not-on-ground <y>)* should be explicitly specified in the effect list since *(on-ground <y>)* is not opposite to *(not-on-ground <y>)* to a machine's percept. To overcome redundancy, IPP or Blackbox use the negative function *not* to negate *P*. Since *not-P* is not used any more, a negative effect can be uniformly handled as *Add(not P)* rather than

as *{Add(not-p), Del(P)}*.

For example, suppose that the *Take-out* operator in a Brief-case domain borrowed from IPP originally has the preconditions, *{in(x), is-at(loc)}*, the add-list, *{not-in(x)}*, and the delete-list, *{in(x)}*. Since the effects of the operator are redundant and can be inferred from the add-list, the operator can be compactly represented as having the preconditions, *{in(x), is-at(loc)}*, the add-list, *{not in(x)}*. Similarly, the *Put-in* operator having the preconditions, *{not-in(x), at(x, loc), is-at(loc)}*, the add-list, *{in(x)}*, and the delete-list, *{not-in(x)}* can be compactly represented without the delete-list.

2.2 Comparison of Two Representations and Opposite Concept

Even though IPP can solve a redundancy problem by using *not*, the advantage of using the negative term comes with its price in terms of the memory space and the time of processing a negation.

Note that while *not-P* is a positive atom, *(not P)* is a composite term. As a matter of fact, it is inconvenient to represent a concept using two words instead of a word, and it is more difficult to handle a negative term than a positive term. We can reasonably hypothesize that it might be more efficient to represent a concept using a positive term rather than using negative complex terms to build an intelligent AI agent.

An aspect of human ingenuity is the ability of avoiding the complexity and the inefficiency by inventing an atomic positive notation to represent a negative concept, and a humans reasoning seems to work according to the principle of Minimum Description Length [4]. We will first conjecture how a human can learn a concept (and its opposite concept) and coin a new positive term to represent a negative concept. A child may learn a concept by dividing a set of objects composed of different properties into subsets of congruent objects consisted of homogeneous property [6]. Suppose a child touches water and learns the concept of *wet* (and *not wet*). Suppose further that *dry* is not yet known to the child. It may be inconvenient to represent a phenomenon each time using *(not wet)*. Similarly, a system using only *not* without new coined terms is limited in its expressive power. We conjecture that a more expressive system should represent the negation of *arm-empty* as *holding(x)* rather than as *(not arm-empty)* in preconditions or effects for operators. It should be pointed out that our approach of using a positive term to represent *(not P)* is limited to a two-valued system, and it is not applicable to a multi-valued system. For example, given a set of attributes, *{white, red, blue, black}*, of a color type in a domain, *(not white)* should mean a disjunctive term *(white ∨ red ∨ blue ∨ black)* rather than *black*.

Note that while *not* is syntactical reasoning, the opposite reasoning is a semantic process. Since a recent system like IPP still cannot understand that *P* and *not-P*, or *on* and *off*, are opposite to each other, an agent equipped with this capability is desirable to be scaled up to a complex dynamic domain. In analogy, similar to the development that *not* is implanted into IPP to expand Graphplan, the opposite function can be implanted into a system to infer a negation, and how to handle opposite function inside an agent should be hidden from a domain theory.

2.3 Empirical Results

To test our hypotheses, we ran a set of planning problems in some IPP domains using the two types of operators, which employ a positive and a negative representation respectively. For example, the Put-in operator in the Briefcaseworld domain with a positive representation has the preconditions {not-in(x), at(x, loc)}, an add-list {in(x)}, and a del-list {not-in(x)}, while the operator in the Negated Briefcaseworld domain with a negative representation has preconditions {not in(x), at(x, loc)} and an add-list {in(x)}. We used the same set of operators to solve the five problems shown below both in Briefcaseworld Domain and in Negated Briefcaseworld Domain. *Ex3a.fct* problem is composed of some objects, such as *paycheck, dictionary, ticket*, etc. In the initial state, the robot (agent) is at home, a paycheck is located at the bank, the dictionary is at the office, and the ticket is at the station. The agent's goal is to move around to bring the objects to home with a briefcase. The other problems are variations of

the first problem with different and somewhat more complicated initial states and / or goals. Even though IPP produces the exactly same plans for the problems in both domains after trying the same number of actions, the processed time and the required memory size was more efficient in the first domain than in the second domain. We measured in the two domains the time spent in terms of *seconds* as shown in the table below:

Name of Problem	Briefcaseworld Domain	Negated Briefcase Domain
Ex3a.fct	0.25	0.37
Ex3b.fct	0.08	0.33
Ex4a.fct	1.86	2.29
Ex5max.fct	3.29	5.28
Ex5d.fct	52.89	68.21

The second table below demonstrates the memory spaces in terms of *KBytes* used to solve the same problems:

Name of Problem	Briefcaseworld Domain	Negated Briefcase Domain
Ex3a.fct	2295.0	3475.7
Ex3b.fct	457.4	808.6
Ex4a.fct	4864.7	7286.2
Ex5max.fct	10848.0	15950.9
Ex5d.fct	10848.0	15950.9

The empirical results shows that using a positive representation of a negative fact is more efficient than using a negative representation both in terms of time and space.

3. Learning Opposite Concept

In the previous section, we observed that a human possesses knowledge unknown to a machine and can immediately detect an inconsistent state. To machine-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. We show that two opposite operators are closely related to mutex in Graphplan. While mutex is inferred syntactically from domain definition, opposite concept is learned from experimentation.

3.1 Machine and Implicit Human Knowledge

The problem of using a negative predicate raises a question of why explicitly encoding control knowledge is necessary to a machine while it is 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 a 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)* $\rightarrow$ *~(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 *~(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 [7]. 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 builds 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 the machine possesses. This type of expert knowledge is called *implicit* knowledge. Since we are not yet at the level of scientifically understanding how the human mind works, especially 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 or as an axiom [8]. However, it is overwhelming to manually encode all the related opposite concepts in a complex domain. Thus, an adaptive intelligent agent should be able to learn an opposite concept autonomously in a new situation.

Suppose that a domain expert does not encode opposite concepts into the domain theory as shown in PRODIGY [9]. Then, while the expert unconsciously uses an opposite concept, a system cannot infer a negative literal. For 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 any current symbolic planning system, 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 concepts can cause unexpected serious problems.

3.2 Finding Opposite Operators

An operator corresponds to an action routine of a robot [5]. Since each routine can be processed independently from other routines, each operator 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 relations 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 if 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 by 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 after 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)* $\not\subset$ *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 satisfy *prestate(op),* and there is an arc from op_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 the preconditions that *op* has deleted. Restoring the preconditions is done by undoing the effects of *op,* that is, by deleting what were added by *add(op)* and adding again what were deleted by *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 can 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. Two actions in Graphplan at the same level are mutex if either 1) the effect of one action is the negation of another actions 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 1) and 3) conditions of the above three conditions, they form opposite operators.

We should point out that there also exist higher-cycles in a complex domain. For example,

consider a variation of the blocksworld domain where the blocks have non-uniform sizes, with super-blocks that can hold two blocks on top of them. Six operators, say, Stack-SuperBl, Stack-LeftBl, Stack-RightBl, Unstack-SuperBl, Unstack-LeftBl, and, Unstack-RightBl, constitute a 6-cycle. The concept of opposite operators does not apply to a higher cycle. However, as Kambhampati et al. [10] state, 2-sized mutexes, which include opposite concepts, are dominant in percentage of action interactions in most classical planning domains with very practical and amazingly successful results, while higher-order interactions between actions as in the super-blocks domain are relatively rare.

3.3 Extracting Opposite Literals

We will show how to extract mutually inconsistent literals from opposite operators using an experimentation method and use them as a rule to learn opposite concept.

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 which 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 an 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 $q_k \rightarrow \sim p_i$. This method is expressed in an algorithm form as follows:

```
Procedure Extract_Opposite_Literals
  Let (op_i, op_j) be opposite operators
  Let add(op_i) ← {p_1, ..., p_n} and add(op_j) ← {q_1,
    ...,q_m}
  for each p_i in {p_1, ..., p_n}  for i=1..n
     Let  S be set to {p_i}
     for each q_k in {q_1, ..., q_n}  for k=1..m
        Let  S' ← S + q_k
        Let same ← Observe_State_Check_Literal
          (S', p_i)
        If same = false  then (p_i, q_k) is opposite
Function Observe_State_Check_Literal (S', p_i)
        return true if p_i is in S'
        else return false
```

For example, suppose *lock-dr* and *unlock-dr* are opposite operators. Let *add(lock-dr)* be *{locked}*, and *add(unlock-dr)* be *{unlocked}*. If S = *{locked}* is an initial state, adding *unlocked* to S, *{locked, unlocked}* is not possible and the state changes to a new state *{unlocked, ~locked}*, thus a rule *unlocked* → *~locked* is learned by experiments. To show a more complicated example containing variables, suppose *stack(x, y)* and *unstack(x, y)* are opposite operators. Let x be instantiated to box_1 and y instantiated to box_2. Let *add(stack(*box_1*,* box_2*))* be *{arm-empty, clear(*box_1*), on(*box_1*,* box_2*)}*, and *add(unstack(*box_1*,* box_2*))* be *{clear(*box_2*), holding(*box_1*)}*. Given that S = *{arm-empty}* is an initial state, adding *clear(*box_2*)* to S, a new state S' = *{clear(*box_2*), arm-empty}* is possible and the two literals are not opposite to each other. On the other hand, adding the next literal *holding(*box_1*)* to S, the expected state *{holding(*box_1*), arm-empty}* is not possible and an unexpected state S' = *{holding(*box_1*), ~arm-empty}* is observed, where *arm-empty* in S is *negated*. Thus, a rule *holding(*box_1*)* → *~arm-empty* is learned by experiments.

If applied as a preprocessor to an incomplete domain theory, this approach of learning opposite function can make an agent more intelligent because a negative literal can be inferred from a positive literal, and the learned rule simplifies a domain theory making an initially incomplete theory more complete and less noisy [11].

4. Conclusion

A planning domain theory represents an agent's knowledge about the task domain. An agent's understanding and manipulating intelligent entities of the domain is a hard but fundamentally important

problem. From observing that a human introduces and utilizes a new predicate, commonly called opposite concept, to negate a given fact based on MDL principle [4], we propose utilizing a new predicate rather than a negative function to represent a negative fact as a technique for building a more intelligent agent, and we show some empirical results in IPP domains supporting our claim in terms of time and space. The opposite operators and propositions in a domain theory are a special type of mutual exclusion first introduced in Graphplan's algorithm. However, the utility of using mutex and *not* is limited to the syntactic aspect of mutual exclusion, and the current efficient planning systems do not understand semantic implication of mutual exclusion [1, 8, 10]. We conjecture that the capacity of understanding a semantically-oriented opposite concept is fundamentally important for an agent to be scaled-up for a real world problem.

References

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

[2] Koehler, J. Extending Planning Graphs to an ADL Subset, Proc. 4th European Conference on Planning, 1997.

[3] Kautz, H. and Salman, B. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search, Proc. of 13th Nat. Conf. AI, 1996.

[4] Rissanen, J. Stochastic Complexity in Statistical Inquiry World Scientific Publishing Company, 1989.

[5] Fikes, R. and Nilsson, N. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, in *Artificial Intelligence* 2, 1971.

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

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

[8] Knoblock, C. Automatically Generating Abstractions for Planning, in *Artificial Intelligence,* 68, 1994.

[9] 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.

[10] Kambhampati, R. and Lambrecht, E, Parker, E. Understanding and extending Graphplan, Proc. 4th European Conference on Planning, 1997.

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

[12] Fox, M and Long, D. The Automatic Inference of State Invariants in TIM, 1998.

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

[14] 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.

[15] Weld, D. Recent Advances in AI Planning, *AI Magazine,* 1999.

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

태 강 수

1983년 전북대학교 영어영문학과 졸업. 1991년 Univ. of North Texas (Computer Science M.S.). 1991년 미국 Texas IBM사 근무. 1997년 Univ. of Texas at Arlington (Computer Eng. Ph.D.). 1997년 성덕대학 전자계산과 전임강사. 1998년 ~ 현재 전주대학교 컴퓨터공학과 조교수. 관심분야는 인공지능, machine learning, planning, internet