In Bramer, M., Preece, A.& Coenen, F. (eds.), Research and Development in Intelligent Systems XVII (Proc. of ES-2000), pp.363-374. Springer-Verlag, 2001.

A Correct Algorithm for Efficient Planning with Preprocessed Domain Axioms

Massimiliano Garagnani
Department of Computing, The Open University
Milton Keynes, U.K.

Abstract

This paper describes a *polynomial* algorithm for preprocessing planning problems which contain domain axioms (DAs) in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow c$. The algorithm presented is an improved version of the (incorrect) transformation for DAs described by Gazen and Knoblock in [6].

The first result presented consists of a counter-example showing that Gazen and Knoblock's preprocessing algorithm is *incorrect*. This is demonstrated by providing a specific set of planning problems which the algorithm does not transform into *equivalent* encodings.

The following result described consists of a new algorithm that avoids the problems of the previous method by augmenting the state with additional assertions ('*deduction facts*') that keep track of any application of the DAs made during the plan.

The final part of the paper illustrates how the new method proposed leads also to notable improvements in the efficiency of the planning process.

1 Introduction

When choosing a language to represent a possible real-world situation, we are confronted with the dilemma of just *how expressive* such language should be. For example, consider a simple blocks world domain, in which four blocks (labeled A,B,C and D in Figure 1) lie on a table. In order to describe this domain it seems natural to include, in the representation language, a predicate expression like `on_top`(x, y), indicating that block x lies on block y. The one-predicate language

$$L = \{ \texttt{on_top}(x, y) \}$$

can actually be used to specify *any* possible state of the system considered[1].

[1] Properties such as `clear`(x) or `on_table`(x) can be *deduced* directly from the current set of `on_top()` propositions.

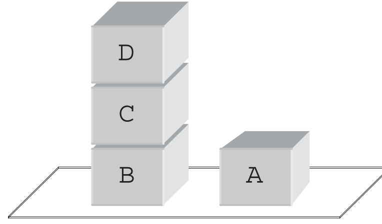

Figure 1: Blocks world domain with four blocks.

However, given an initial random disposition of the blocks, suppose our goal g to consist of building a pile of four blocks having 'A' on top. If we adopt the language L to describe this problem, we will have to express g as a six-term disjunctive expression, in which every term represents one of the possible permutations of the blocks B,C,D beneath A:

$$\begin{aligned} g \quad = \quad & (\texttt{on_top}(\texttt{A},\texttt{B}) \wedge \texttt{on_top}(\texttt{B},\texttt{C}) \wedge \texttt{on_top}(\texttt{C},\texttt{D})) \\ & \vee(\texttt{on_top}(\texttt{A},\texttt{B}) \wedge \texttt{on_top}(\texttt{B},\texttt{D}) \wedge \texttt{on_top}(\texttt{D},\texttt{C})) \\ & \vee(\texttt{on_top}(\texttt{A},\texttt{C}) \wedge \texttt{on_top}(\texttt{C},\texttt{B}) \wedge \texttt{on_top}(\texttt{B},\texttt{D})) \\ & \vee(\texttt{on_top}(\texttt{A},\texttt{C}) \wedge \texttt{on_top}(\texttt{C},\texttt{B}) \wedge \texttt{on_top}(\texttt{B},\texttt{D})) \\ & \vee(\texttt{on_top}(\texttt{A},\texttt{D}) \wedge \texttt{on_top}(\texttt{D},\texttt{B}) \wedge \texttt{on_top}(\texttt{B},\texttt{C})) \\ & \vee(\texttt{on_top}(\texttt{A},\texttt{D}) \wedge \texttt{on_top}(\texttt{D},\texttt{C}) \wedge \texttt{on_top}(\texttt{C},\texttt{B})) \end{aligned}$$

To avoid this exponential growth in complexity of goals and conditions, we can *extend* the language L to include more expressive terms. For example, if we augment L with the predicate '$\texttt{above}(x,y)$', indicating that the block 'x' is on the same pile of 'y' but in a higher position, the goal g can be re-stated simply as

$$g = \texttt{above}(\texttt{A},\texttt{B}) \wedge \texttt{above}(\texttt{A},\texttt{C}) \wedge \texttt{above}(\texttt{A},\texttt{D})$$

The new predicate can be defined unequivocally through the following pair of *domain* (or *language*) *axioms* (*DAs*):

$$\begin{aligned} \texttt{on_top}(x,y) \quad & \rightarrow \quad \texttt{above}(x,y) \\ \texttt{on_top}(x,y) \wedge \texttt{above}(y,z) \quad & \rightarrow \quad \texttt{above}(x,z) \end{aligned}$$

This means that the truth of any '`above()`' expression can actually be *deduced* from the set of '`on_top()`' expressions currently holding.

The adoption of an expressive language allows a simple, accurate and natural description of the problem considered. Nevertheless, the gain in simplicity and clarity of the problem definition language is usually counterbalanced by a loss in performance. In other words, the presence

of axioms in the domain makes, in general, the automatic solution of problems more complex.

This is often the case in the field of AI Planning, where systems that support very expressive domain definition languages (such as UCPOP [8], which allows the use of DAs) are generally slower in finding a plan solution than other planners adopting more restricted languages (e.g. [1] [7]).

A possible approach to this seemingly contradictory situation consists of developing *preprocessors* that translate domains from an expressive representation language into a simpler one for which more efficient planners exist. More specifically, given a planning problem containing domain axioms, the preprocessor will transform it into a new planning problem *equivalent*[2] to the original one but containing no DAs. The new problem can then be solved by a fast planner, and the solution found re-transformed into an equivalent plan which solves the initial problem.

This is the approach that Garagnani adopted in [5] to preprocess *discourse planning* problems containing *acyclic* belief axioms (see also [4]). The same approach has been followed by Gazen and Knoblock in [6], where an algorithm for the conversion of a UCPOP domain representation into a Graphplan [1] (equivalent) encoding is presented. The algorithm described includes the transformation of domain axioms into equivalent 'deduce' operators. As the authors point out, the advantages of this approach reside in its conceptual simplicity, its modularity and the fact that it is not necessarily specific to one planner (cf. [6, p.222]).

The first part of this paper will show that Gazen and Knoblock's transformation of DAs is *incorrect*. In other words, the planning problem resulting as output of their preprocessing algorithm is *not always equivalent* to the initial problem which was given as input.

In the second part of the paper, a revised version of the algorithm for the preprocessing of DAs is presented. The new method avoids the problems of the previous solution by augmenting the state with extra assertions (called '*deduction facts*') which keep track of *any* use of the axioms made during the plan.

Finally, some preliminary results showing how the output produced by the new transformation leads also to more efficient planning are presented.

[2]Two planning problems $\mathcal{P}, \mathcal{P}'$ can be considered 'equivalent' if there exists a bijective function which associates every solution (successful plan) of $\mathcal{P}$ with one (and only one) solution of $\mathcal{P}'$, and vice versa.

2 Gazen and Knoblock's algorithm

This section reviews the preprocessing algorithm presented by Gazen and Knoblock in [6] and demonstrates its incorrectness.

The algorithm described in [6, p.225] for the transformation of DAs in the form *premises* → *consequence*[3] into 'deduce' operators is reported below:

ALGORITHM α

Given: an axiom $p \rightarrow c$ and a list of operators l_o, let o be a new operator:

1) set the precondition of o to p
2) set the effect of o to c
3) for each op in l_o
    if `effect(`op`)` contains any predicate in p,
    then add $(\forall(y_1 \ldots y_n)$ `(not` $(c\ y_1 \ldots y_n)$`)` $)$ to `effect(`op`)`
4) add o to l_o

The reason for the introduction of step 3) in the above algorithm is that if a step in the plan solution modifies one of the components of the premise p of an axiom, the proposition c deduced by the operator o may lose its validity. Hence, according to Gazen and Knoblock, "it is necessary [during the preprocessing phase] to find the operators that modify the propositions from which the axiom is derived, and to add an effect which negates the deduced proposition" [*ibid.*, p.224].

However, the modification of the premises of a specific axiom does not require to *unconditionally* delete *every instance* of the axiom's consequence. This is, in fact, the origin of the algorithm's incorrectness. The following subsection describes an example in which this weakness is exploited to construct a situation in which the planning problem given as input to Algorithm α is *not equivalent* to the output produced.

2.1 Proof of *in*correctness

In order to prove that Algorithm α is not correct, it is sufficient to provide a counter-example in which the input planning problem is not transformed into an equivalent one. Consider the following problem $\mathcal{P}$, consisting of two operator schemes (represented, as in STRIPS [3], like

[3] '*premises*' is an expression which can contain predicates and which must be true before the axiom can be applied, whereas '*consequence*' is a single predicate c with n arguments $(c\ x_1 x_2 \ldots x_n)$.

triples (P,A,D) of preconditions, add and delete lists), initial state I, goal set G and single domain axiom A_1:

$$\begin{array}{rcll} Op_1(x) & = & ([Q],[C(x),S],[Q]) & x \in \{a,b,c,\ldots,w,z\} \\ Op_2(x) & = & ([S,P(x)],[R(x)],[S,P(x)]) & x \in \{a,b,c,\ldots,w,z\} \end{array}$$

$$\begin{array}{rcl} I & = & \{Q,P(b)\} \\ G & = & \{C(a),R(b)\} \end{array}$$

$$A_1) \qquad P(x) \rightarrow C(x) \qquad x \in \{a,b,c,\ldots,w,z\}$$

The given problem has one (and only one) solution, shown in Figure 2. Notice that the solution does not make use of the axiom A_1.

The planning problem $\mathcal{P}'$ produced by the preprocessing algorithm α is:

$$\begin{array}{rcll} Op_1{}'(x) & = & ([Q],[C(x),S],[Q]) & x \in \{a,b,c,\ldots,w,z\} \\ Op_2{}'(x) & = & ([S,P(x)],[R(x)], & \\ & & \quad [S,P(x),\forall y \overset{Del}{\Rightarrow} C(y)]) & x,y \in \{a,b,c,\ldots,w,z\} \end{array}$$

$$\begin{array}{rcl} I' = I & = & \{Q,P(b)\} \\ G' = G & = & \{C(a),R(b)\} \end{array}$$

$$O_1(x) \; = \; ([P(x)],[C(x)],[\,]) \qquad x \in \{a,b,c,\ldots,w,z\}$$

Due to the presence of the *unconditional* deletion in $Op_2{}'$, the plan $\langle Op_1{}'(a),\, Op_2{}'(b)\rangle$ would now produce the state $S_2' = \{R(b)\}$, which is not a goal state. As a matter of fact, the problem $\mathcal{P}'$ produced by the algorithm α presents *no solution* at all, and therefore is not equivalent to the initial planning problem $\mathcal{P}$. Hence, the preprocessing algorithm proposed by Gazen and Knoblock is incorrect.

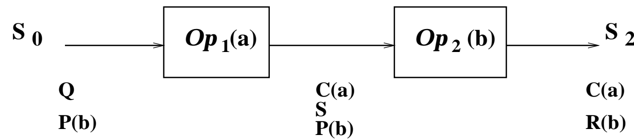

Figure 2: Correct plan solution.

3 The new algorithm

This section describes a revised version of Gazen and Knoblock's algorithm α for the transformation of domain axioms into equivalent 'deduce'

operators. The new algorithm, reported below, avoids the previous problems by producing operators that effect *conditional* deletions based on '*deduction facts*' – assertions in the form $A(\vec{v})$ [4] which do not belong to the definition language L. The task of the deduction facts is to record *all instances* of application of the DAs, so that if a premise p of an axiom is deleted by an operator, *all (and only)* the consequences of p can be identified and removed from the state:

ALGORITHM β

Given: a list of domain axioms $\{A_1, A_2, \ldots, A_l\}$ in the form

$$
\begin{array}{l}
A_i)\ p_{i,1}(\vec{x}_{i,1}) \wedge p_{i,2}(\vec{x}_{i,2}) \wedge \ldots \wedge p_{i,k_i}(\vec{x}_{i,k_i}) \rightarrow c_i(\vec{y}_i) \\
\vec{x}_{i,j} = (x_{i,1}, x_{i,2}, \ldots, x_{i,m_j}) \\
\vec{y}_i = (y_{i,1}, y_{i,2}, \ldots, y_{i,n_i})
\end{array}
$$

and a list of operators l_o in the form (P,A,D),

- for each axiom A_i, let O_i be a new deduce operator:

  1) set the preconditions P of O_i to $\{p_{i,1}(\vec{x}_{i,1}), \ldots, p_{i,k_i}(\vec{x}_{i,k_i})\}$
  2) set the add list A to $\{c_i(\vec{y}_i)\}$
  3) append the *conditional effect* $[\neg c_i(\vec{y}_i) \stackrel{Add}{\Rightarrow} A_i(\vec{x}_{i,1}, \vec{x}_{i,2}, \ldots, \vec{x}_{i,k_i}, \vec{y}_i)]$
  4) for each premise p_{ij} that appears also as *consequence* c_w of an axiom A_w, append to O_i the conditional effect
     $\forall r, s \ldots t\ [A_w(r, s, \ldots, t, \vec{x}_{i,j}) \stackrel{Add}{\Rightarrow} A_i(r, s, \ldots, t, \vec{y}_i)]$

  5) for each Op=(P,A,D) in l_o,

     5.1) if D contains a premise $p_{i,j}$,
     then append to Op the conditional deletion
     $\forall r, s, \ldots t\,[A_i(r, s, \ldots, \vec{x}_{i,j}, \ldots, t, \vec{y}_i) \stackrel{Del}{\Rightarrow} c_i(\vec{y}_i)]$

     5.2) if A contains the consequence c_i,
     then for each axiom A_w with consequence c_i, append
     $\forall r, s, \ldots t\,[\emptyset \stackrel{Del}{\Rightarrow} A_w(r, s, \ldots, t, \vec{y}_i)]$

- add $\{O_1, O_2, \ldots, O_l\}$ to l_o

[4] $\vec{v}$ is a tuple $(r, s, \ldots, t)$ of variable length.

———

Perhaps the first thing to notice is that the complexity of this transformation is *polynomial* in the number 'l' of DAs and in the cardinality of the initial list l_o of operators. As in [6], for each axiom A_i — with premises p and consequence c — a new deduce operator O_i = (P,A,D) = $([p],[c],[\,])$ is generated. However, the operator is also augmented with the (conditional) addition of a deduction fact $A_i(\,)$ (see step 3) which records unequivocally the *bound values* of the variables in A_i at the moment of its application. This will allow the identification of exactly *which premises* have led to *which consequence* at any subsequent point in the plan.

In what follows, the functioning of the Algorithm β is explained through a detailed example.

3.1 An explanatory example

Consider the blocks world definition language L of the example in Section 1 enriched with the term '$\texttt{under}(x,y)$', indicating that the block 'x' is on the same pile of 'y' but in a lower position. The extension of the expressiveness of L requires the addition of the two following DAs:

$$\begin{array}{lll} A_1) & \texttt{on_top}(x,y) & \rightarrow \texttt{under}(y,x) \\ A_2) & \texttt{on_top}(x,y) \wedge \texttt{under}(z,y) & \rightarrow \texttt{under}(z,x) \end{array}$$

The following subsection illustrates how steps 1)–4) of the β algorithm transform the above DAs into equivalent deduce operators.

3.1.1 Steps 1)–4)

Consider the axiom A_1 first. The steps 1)–3) of β will initially produce the following deduce operator O_1 (represented as in [7]):

$$\begin{array}{rcl} \mathrm{P} & = & [\texttt{on_top}(x,y)] \\ \mathrm{A} & = & [\texttt{under}(y,x)] \\ \mathrm{D} & = & [\,] \\ \textit{Effects}: & & [\neg\texttt{under}(y,x) \overset{Add}{\Rightarrow} \texttt{A}_1(x,y,y,x)] \end{array}$$

Notice that the addition of the deduction fact $\texttt{A}_1(x,y,y,x)$ is subject to the condition $\neg\texttt{under}(y,x)$. This guarantees that the 'tag' $\texttt{A}_1(x,y,y,x)$ is added to the state only when the term $\texttt{under}(y,x)$ is present *solely* as a consequence of the premise $\texttt{on_top}(x,y)$.

Step 4) of the algorithm does not add any effect to O_1. Let us postpone, for the moment, the execution of step 5), and move on to the analysis of the transformation of axiom A_2. The operator O_2 resulting from the first 3 steps of β is shown below:

$$\begin{aligned} \text{P} &= [\texttt{on_top}(x,y), \texttt{under}(z,y)] \\ \text{A} &= [\texttt{under}(z,x)] \\ \text{D} &= [\,] \\ \textit{Effects}: &\quad [\neg\texttt{under}(z,x) \overset{\textit{Add}}{\Rightarrow} \texttt{A}_2(x,y,z,y,z,x)] \end{aligned}$$

In this case, one of the premises of the axiom — namely, $\texttt{under}(z,y)$ — appears as consequence in other DAs. This means that the corresponding precondition of the operator could have been *derived* from other premises through the application of deduce operators. If any of such premises is deleted, then $\texttt{under}(z,y)$ should be removed, and so should $\texttt{under}(z,x)$. Hence, the premises which led to the derivation of $\texttt{under}(z,y)$ should be considered also as premises of $\texttt{under}(z,x)$. This is the reason for the introduction of step 4), which causes the addition of the following conditional effects to the deduce operator O_2:

$$\begin{array}{lll} \forall r,s & [\texttt{A}_1(r,s,z,y) & \overset{\textit{Add}}{\Rightarrow} \texttt{A}_2(r,s,z,x)] \\ \forall r,s,t,u & [\texttt{A}_2(r,s,t,u,z,y) & \overset{\textit{Add}}{\Rightarrow} \texttt{A}_2(r,s,t,u,z,x)] \end{array}$$

Having analysed the output that β produces in steps 1) to 4) for the two DAs, let us now move on to step 5), which modifies the initial set l_o of operators.

3.1.2 Step 5)

Step 5.1) concerns the deletion of one of the premises $p_{i,j}$ appearing in the left-hand side of an axiom. Such deletion will cause the removal of all (and only) the consequences which have been *directly or indirectly* derived from it. These consequences can be easily identified from the set of recorded deduction facts containing $p_{i,j}$ (or, rather, its *ground instance*) as one of the premises.

In the example considered, if an operator contains the term $\texttt{on_top}(x,y)$ in its delete list D, then it will be augmented with the following conditional deletions:

$$\begin{array}{lll} \forall t,u & [\texttt{A}_1(x,y,t,u) & \overset{\textit{Del}}{\Rightarrow} \texttt{under}(t,u)] \\ \forall r,s,t,u & [\texttt{A}_2(x,y,r,s,t,u) & \overset{\textit{Del}}{\Rightarrow} \texttt{under}(t,u)] \end{array}$$

These effects will remove all and only the consequences '$\texttt{under}(t,u)$' which have been derived (directly or indirectly) from the term '$\texttt{on_top}(x,y)$' through the application of axioms A_1 and A_2.

Finally, step 5.2) is necessary in order to guarantee that any deduction 'tag' $\texttt{A}_i(\vec{v})$ is present in the state *iff* the consequence it contains has been added *exclusively* because of the application of the axiom A_i. In fact, if one of the operators in the list l_o *explicitly adds* a proposition c (i.e., $c \in$ A), such term should be no longer considered as 'derived' from others, and *all* of the deduction facts containing c as a consequence should be deleted.

In the example, if an operator Op contains the term $\texttt{under}(x,y)$ in its add list A, then Op will be augmented with the following (unconditional) deletions:

$$\begin{array}{lllll} \forall r,s & [\emptyset & \overset{Del}{\Rightarrow} & \texttt{A}_1(r,s,x,y)] \\ \forall r,s,t,u & [\emptyset & \overset{Del}{\Rightarrow} & \texttt{A}_2(r,s,t,u,x,y)] \end{array}$$

3.2 Correctness of β

In order to show that the new algorithm does not present the same problems of Gazen and Knoblock's version, let us consider the counter-example of Section 2.1. Such example was built explicitly to prove that Algorithm α was incorrect, as not always producing a planning problem equivalent to the one given as input. The output $\mathcal{P}''$ of the transformation β applied to the planning problem $\mathcal{P}$ of the example is reported below:

$$\begin{array}{rcl} Op_1{}''(x) & = & (\,[Q],[C(x),S],[Q,A_1(x,x)]\,) \\ \\ Op_2{}''(x) & = & (\,[S,P(x)],[R(x)],[S,P(x)]\,) \\ \textit{Effects}: & & \forall y\,[A_1(x,y) \overset{Del}{\Rightarrow} C(y)]) \\ \\ I'' & = & \{Q,P(b)\} \\ G'' & = & \{C(a),R(b)\} \\ \\ O_1(x) & = & (\,[P(x)],[C(x)],[\,]\,) \\ \textit{Effects}: & & [\neg C(x) \overset{Add}{\Rightarrow} A_1(x,x)]) \end{array}$$

The execution of the plan $\langle Op_1{}''(a), Op_2{}''(b)\rangle$ yelds now the correct goal state S_2 of Figure 2. This is because the conditional deletion in $Op_2{}''(x)$ has been *limited* to the terms $C(y)$ for which $A_1(x,y)$ holds.

Hence, the output produced by β consists of a planning problem $\mathcal{P}''$ having one and only one solution, that is, *equivalent* to the problem $\mathcal{P}$ given as input.

4 Planning performance and preliminary results

Although the algorithm α has been shown to be incorrect in a specific case, one might argue that such situation was created 'artificially', and will never occur in real-world problems. Nevertheless, not only is Gazen and Knoblock's algorithm incorrect, but it also produces problem encodings that lead to *inefficiencies* during the planning process. In fact, consider the following example, taken from the original paper [6, p.225]. Given

- Axiom `is-clear`: (or (eq x Table) ($\nexists$ (obj b) (on b x))) $\rightarrow$ (clear x)

- Operator `put-on`(x, y, d):
  Precondition = (and (on x d) (clear x) (clear y))
  Effect = (and (on x y) (not (on x d)))

the preprocessing algorithm α would transform them into:

- Operator `deduce-is-clear`(x):
  Precondition = (or (eq x Table) ($\nexists$ (obj b) (on b x)))
  Effect = (clear x)

- Operator `put-on`(x, y, d):
  Precondition = (and (on x d) (clear x) (clear y))
  Effect = (and (on x y) (not (on x d)) ($\forall$ (v) (not (clear v))))

The effect '$\forall$(v) (not (clear v))' deletes every occurrence of the proposition (clear v), regardless of which instance of '`deduce-is-clear`' had actually produced it. This leads to inefficiencies during the planning process, as the unconditional deletion forces *any* '(clear v)' proposition to be re-evaluated subsequently if needed by another operator.

Gazen and Knoblock are well aware of this problem: "in the worst case an axiom may need to be asserted after each step". For example, the '`put-on`' operator above "does not have to assert (not (clear x)) because x is still clear after this action, but because it does, the axiom needs to be applied to re-assert (clear x) if another action requires (clear x) later" [*ibid.*,p.225].

In contrast, the new method presented is correct and avoids the above kind of inefficiencies by keeping track of each application of the deduce operators, so that when the premise of an axiom is removed, only the propositions which were actually *derived* from it are deleted.

Results showing a comparison between the performances of the UCPOP planner [8] and those obtained using the preprocessing algorithm α in conjunction with Graphplan [1] are discussed in [6]. In order to have at least a preliminary quantification of the possible gain in efficiency produced by the adoption of the new algorithm, the planning problem reported below (Figure 3) with axioms A_1, A_2 and language L — in fact, a variation of Sussman anomaly [2] — was preprocessed with both the α and β versions varying the number of blocks present in the initial pile, and the results produced were given as input to the IPP planner [7].

$$L = \{\texttt{on_top}(x,y), \texttt{on_table}(x), \texttt{clear}(x), \texttt{above}(x,y)\}$$

$$\begin{array}{lllll} A_1) & \texttt{on_top}(x,y) & & \rightarrow & \texttt{above}(x,y) \\ A_2) & \texttt{on_top}(x,y) \wedge \texttt{above}(y,z) & & \rightarrow & \texttt{above}(x,z) \end{array}$$

UNSTACK(x,y,z)

P	above(x,z), on_top(x,y), clear(x), on_table(z)
A	on_table(x), clear(y)
D	on_top(x,y)

STACK(x,y,z)

P	above(y,z), on_table(z), on_table(x), clear(x), clear(y)
A	on_top(x,y)
D	clear(y), on_table(x)

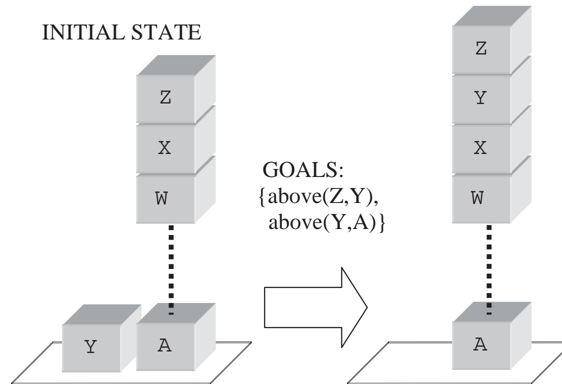

Figure 3: Blocks world planning problem(s) used for comparing α and β.

It should be noticed that the operator schemes `Stack` and `Unstack` were deliberately adopted instead of the more common `Pick-up`, `Put-down` and `Put-on` in order to obtain a situation in which their application were always preceded by a sequence of 'deduction' steps. Such behaviour, in fact, is guaranteed by the presence of the term '`above( )`' in their preconditions. As a result, although the solution to the given problem(s) always required essentially the three steps `Unstack`(Z,X,A), `Stack`(Y,X,A) and `Stack`(Z,Y,A), the plans obtained with the output of the β algorithm were found to be roughly 75% *shorter* than those obtained with the α version. Such difference was due to the fact that both of the α versions of the operators removed *all* the '`above( )`' terms from the state. This forced the plan to re-assert them all again after each step in order to re-deduce the required '`above( )`' preconditions and, at the end, the final goals.

5 Discussion and Future work

The advantages of adopting a preprocessing approach to the problem of planning with DAs lie in its modularity and in its wide range of applicability. More specifically, the solution proposed is correct, more efficient than the existing version and not tied to any particular planner.

A possible extension of the algorithm presented consists of modifying it in order to deal with DAs containing *existentially* and *universally* quantified variables. This would be required to transform the example reported in Section 4. An extended version of β able to deal with axioms containing negated existentials has been already developed and is currently under testing.

Another direction in which work is in progress consists of using the deduction facts added to the state to automatically *learn new* operators and axioms. This can be done during the planning process through '*constant-variable*' substitution[5] or, in alternative, by exploiting the set of deduction facts `A`$(\ldots, c)$ present in the state. In fact, at any point in the plan, the set of deduction facts containing a consequence 'c' identifies unequivocally the set of premises which have been used to conclude c. Hence, for each axiom it is possible to introduce in the problem a new operator that *learns* from past 'chunks' of the plan and improves the efficiency of the subsequent planning process. A possible version of such a learning operator for an axiom A_x with premises $p_1, p_2, \ldots, p_n$ and

[5] A collection of axioms is generalised and grouped into a single deduce operator by substituting identical constants with identical variables.

consequence c is proposed below:

$$\begin{aligned} \mathrm{P} &= [\forall \vec{r}, \vec{s}, \ldots, \vec{t} \cdot \mathtt{A}_x(\vec{r}, \vec{s}, \ldots, \vec{t}, \vec{y}),\ p_1(\vec{r}) \wedge p_2(\vec{s}) \wedge \ldots \wedge p_n(\vec{t})] \\ \mathrm{A} &= [c(\vec{y})] \\ \mathrm{D} &= [\,] \end{aligned}$$

In brief, regardless of the *intermediate steps* which led to the deduction of the consequence c (and which may be 'forgotten'), the list of deduction facts $\mathtt{A}_x(\,)$ identifies unequivocally the collection of premises that have been progressively used to derive 'c'. Hence, if all of them hold again, then c can be added to the state.

In summary, the automated translation from a high-level domain definition language into a simpler one makes it possible to use simple but fast planners in complicated domains. The gain in planning performance is achieved by making the information initially encoded within the DAs and the operators more *explicit* in the problem description. Having axioms in a domain definition language allows the domain engineer to represent problems easily, accurately and naturally, and leads to 'cleaner' and less error-prone problem definitions. In a way, preprocessing undoes that, and although the resulting problem is not as compact, its correctness is preserved (work is in currently progress to produce a formal proof of the soundness of the preprocessing algorithm β).

In conclusion, the preprocessing of problems containing domain axioms and, more in general, of complex real-world situations defined using expressive languages seems to represent a valid method for the construction of efficient AI systems, able to learn and support high-level, natural domain-definition languages.

References

[1] Blum, A. L., Furst, M. L. (1997) "Fast Planning Through Planning Graph Analysis", *Artificial Intelligence*, 90:281–300.

[2] Chapman, D. (1987) "Planning for Conjunctive Goals", *Artificial Intelligence*, 32(3):333-377.

[3] Fikes, R.E., Nilsson, N.J. (1971) "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, 2:189-208.

[4] Garagnani, M. (2000) "Speaker-hearer beliefs for discourse planning", *Proceedings of the 17th International Conference on Artificial Intelligence (IC-AI'00)*, Las Vegas, Nevada, June 2000.

[5] Garagnani, M. (1999) "A sound Linear Algorithm for Preprocessing planning problems with Language Axioms", *Proceedings of PLANSIG-99*, Manchester, England.

[6] Gazen, B.C., Knoblock, C.A. (1997) "Combining the Expressivity of UCPOP with the Efficiency of Graphplan", *Proceedings ECP-97*, Toulouse, France.

[7] Koehler, J., Nebel, B., Hoffmann, J., Dimopoulos, Y. (1997) "Extending Planning Graphs to an ADL Subset", *Proceedings ECP-97*, Toulouse, France.

[8] Penberthy, J. S., Weld, D. (1992) "UCPOP: A Sound, Complete, Partial-order Planner for ADL", *Proceedings of the International Workshop on Knowledge Representation (KR-92)*, pp.103–114.