Learning Plan Rewriting Rules

M. Afzal Upal
Faculty of Computer Science
Dalhousie University
Halifax, NS, B3H 1W5
Canada
email: afzal.upal@dal.ca

Abstract

Considerable work has been done to automatically learn domain-specific knowledge to improve the performance of domain independent problem solving systems. However, most of this work has focussed on learning search control knowledge—knowledge that can be used by a problem solving system during search to improve its performance. An alternative approach to improving the performance of domain independent systems is by using rewriting rules. These are the rules that can be used by a problem solving system after generating an initial solution in order to transform it into a higher quality solution. This paper reviews various approaches that have been suggested for automatically learning rewriting rules, analyses them, and suggests novel algorithms for learning plan rewriting rules.

The ability to produce high quality plans is essential if AI planners are to be applied to the real world planning problems. Machine learning for planning suggests automatically learning domain specific information that can be used by the AI planners to produce high quality solutions. Considerable planning and learning research has been devoted to learning domain specific search control rules to improve planning performance (planning efficiency and plan quality). These rules improve planning performance by guiding a planner towards higher quality plans during planing. An alternative technique is planning by rewriting (Ambite & Knoblock 1997) that suggests first generating an initial plan using a planner and then using a set of rewrite-rules to transform it into a higher quality plan. However, automatically learning such rules has been a challenging problem. Previously, I designed a system called REWRITE (Upal 1999; 2000) that automatically learned plan rewrite rules by comparing two planning search trees (one search tree leading to a lower quality plan and the other search tree leading to a higher quality plan). However, REWRITE's performance with respect to planning efficiency was poor (Upal & Elio 2000) because it was unable to learn good rewrite rules. In (Upal 2000) we show that the problem lies with REWRITE's strategy of learning rules in one context (i.e., the context of search nodes) and then applying them in a different context (i.e., the context of the complete plans). (Upal 2000) also presents evidence to show that learning search control rules from search trees is a better option than learning rewrite rules by comparing planning traces.

Ambite et al. (Ambite, Knoblock, & Minton 2000) suggest a similar technique for learning rewrite rules. However, their system (called Pbr) learns by comparing two completed solutions to a planning problem; one of higher quality and other of lower quality. Pbr's learning component simply stores those steps of the better quality solution that are not present in the lower quality plan as replacing steps, and it stores those steps of the lower quality solution that are not present in the higher quality solution as the to-be-replaced steps and to-be-replaced constraints on the steps. Pbr also ranks its rules according to goodness. It prefers the rules that are smaller.

Next I present the background material on planning by rewriting followed by a review of the previously suggested algorithms and suggest new algorithms for learning plan rewriting rules.

Planning by Rewriting

The basic idea of rewriting can be traced back to the work on graph and number rewriting (Baadr & Nipkow 1998). The idea of using rewrite rules for AI planning was introduced by Ambite et al. (Ambite & Knoblock 1997) who referred to it as planning by rewriting.

A planning by rewriting system consists of two component: a planning component for generating the initial plans, and a rewriting component that can rewrite these plans to transform them into higher quality plans. Since, by definition, planning by rewriting systems spend extra time in plan-rewriting, planning by rewriting can be expected to be useful in those planning domains in which generating suboptimal plans is significantly more efficient than generating optimal quality plans. Interestingly, many AI planning domains such as Blocksworld and the logistics transportation domain exhibit these properties (Ambite & Knoblock 1997).

A rewrite rule consists of two equivalent sequences of actions such that one of them can be replaced by the
Figure 1: A rewrite rule for the logistics transportation domain.

Figure 2: (a) An initial suboptimal plan from logistics transportation domain. (b) The rewritten plan.

Good & Bad Rewrite Rules
Unfortunately, the new graph obtained by applying the rewrite rule may no longer correspond to a viable plan. For instance, when the rewrite rule of Figure 1 is applied to planning graph of Figure 2(a) the subgoal at-truck(mygm, apt) becomes unresolved. Some re-planning has to be done to transform this incomplete plan into a complete plan again. Clearly, the more the re-planning that is required the less efficient a planning by rewriting system will be. Hence, a rewrite rule learner must:

- learn rewrite rules that can be used efficiently by the plan rewriting process (i.e., the rules that require little re-planning effort after their application), and
- use an efficient plan rewriting process.

For instance, consider the rules that consist of two action sequences that have exactly the same available effect set (the set of effects that actions in a subsequence can supply to the outside actions) and the same net precondition set (the set of unresolved goals that an action subsequence has). Using such rules will require minimal re-planning effort because re-planning is guaranteed to lead to complete plans without adding any new actions. Unfortunately, such rules may be too specific to be used in a broad set of situations. In general, the bias for efficient re-planning can be expected to favor over-specific rewrite rules.

Rewriting efficiency, however, is not the only criteria for determining the goodness of a plan rewriting process (in particular, for a rewriting module that is to be used as a part of a learning and planning system). From a machine learning perspective, a learning system must be able to learn general rules that are effective in leading to performance improvements in a broad set of situations. Hence a good rewrite rule learning system must:

- learn rewrite rules that are effectively applicable in a large number of situations, and
- use an effective rewriting process (i.e., its re-planning component is able to successfully complete as many plans made incomplete by the rewrite rule application as possible).

For instance, consider a rewrite rule learner that generates rules by placing all subsets of the set of domain actions in its to-be-replaced and replacing action sequences. Such rules will guarantee generation of optimal quality plans. The problem is that many of these rules will be applicable in too broad a set of situations but may only lead to performance improvements in a

1If we also assume that the to-be-replaced and the replacing actions are ordered then no re-planning is required because the new plan can simply be obtained by substituting the replacing actions in place of the to-be-replaced actions.

2For instance, rules that have an empty set as a to-be-replaced action sequence will be applicable in every situation.
small subset of them. In general, the bias for large gains in the learning performance can be expected to favor over-general rewrite rules.

Finding the right balance between over-specific and over-general rewrite rules is a challenging machine learning problem. However, little work has been done to compare the consequences of different design decisions in the context of rewrite rules. Most of the work has focussed on finding a way to learn rewrite rules automatically. Ambite's Pbr (Ambite & Knoblock 1997) used a local search technique for rewriting. We implemented two versions of REWRITE called REWRITE-first and REWRITE-best using the first and the best rewriting strategies respectively (Upal & Elio 2000). The first rewriting strategy only performs one rewriting of an initial plan whereas the best rewriting strategy performs all possible rewritings to return the best quality plan found.

Our experiments with the two rewriting showed that if a rewrite rule learner is not selective with regards to size of its rewrite rule library then even the first rewriting strategy becomes too inefficient (Upal & Elio 2000). Ambite et al. (Ambite, Knoblock, & Minton 2000) suggest that a rewrite rule learner should prefer smaller rules because they are more likely to have a good balance of specificity and generality. However, we believe that this criteria is too simple because it does not take into account factors such as the gain in plan quality that is expected to be achieved as a result of applying the rule. We suggest a broader criteria for measuring the goodness of a rewrite rule:

1. gain in plan quality that is expected to be achieved as a result of a successful application of the rewrite rule.
2. Context similarity of the to-be-replaced actions sequence with the replacing action sequence. This can be measured by measuring the similarity between the precondition and the effect sets of the two actions sequences.
3. Simplicity of the rewrite rule. This can be measured by measuring the size of the rewrite rule. The smaller a rewrite rule the better it is expected to be.

Good & Bad Rewriting Processes

Similar to the conflict between over-general and over-specific rewrite rules, a conflict also exists between efficient and effective rewriting processes. For instance, consider the following two rewriting processes.

1. A rewriting process that uses a partial-order planner as a plan rewriting procedure.
2. A rewriting process that uses a simplified partial-order planner that can only use establishment to resolve open condition flaws as its replanning procedure.

Using the second rewriting algorithm the process of determining whether an incomplete plan can be rewritten or not can be performed more efficiently because it has strictly fewer number of choice points and hence a smaller search space to explore. However, such a replanning process would be unable to successfully replan (and hence improve plan quality of) a larger number of plans. Therefore, it will be less general than the first rewriting process.

Another variable in a rewriting system is the number of ways the initial plan can be rewritten. The reason is that a number of rules may be applicable to a plan. Application of each of these rules may lead to a number of different rewritten plans of different quality. This number can be as large as the number of ways of applying (i.e., instantiating) all the applicable rewrite rules. The benefit of applying all rewrite rules is that it allows evaluation of the entire neighborhood and hence the best quality plan can be obtained. However, searching the entire neighborhood can be inefficient. If we restrict the ways of rewriting a plan to the first feasible way of rewriting, then the rewrite algorithm becomes efficient. The drawback is that the rewriting system is not making use of all the learned knowledge and hence is not as effective as it can be.

In summary, the challenge for a rewrite rule learner is to learn plan rewriting rules that are:

- not too general or too specific, and are
- efficient, and
- effective.

Next section compares a number of rewriting rule learning algorithms with respect to the criteria of how efficiency and effectiveness.

Algorithms for Learning Rewrite Rules

Learning Through Static Domain Analysis

Recently, there has been a surge of interest in automatically learning domain specific rules by statically analyzing planning operators and/or the problem descriptions without solving the planning problems (Fox 2000). Even though most of this work has focussed on learning search control rules, such analysis can also be used to learn rewrite rules (for instance, by creating all possible combinations of domain actions in to-be-replaced and replacing actions). A number of improvements to this algorithm are possible. Figure 3 shows Algorithm 1 which is based on a number of heuristics that we have found to be useful for discovering good rewrite rules quickly.

Step 1 of the Algorithm 1 compares all domain actions with one another to find the ones that have an effect in common. If they do then it creates a rewrite rule with the more costly action as the to-be-replaced part of the rule and the less costly action as the replacing part of the rule and adds this rule to the set of rules found so far and tries to add an action to the chain by searching for an action that can supply a precondition of an action that is already in the chain. Every time it adds an action to a chain, it stores the chain with a higher quality plan in the replacing part of the
**algorithm 1:** (Domain-operator-set)

Set-of-Rules ← \{\}\nfor all subsets \(s_i = \{a_1, a_2\}\) of size 2 of Domain-operator-set do
if \(a_1\) and \(a_2\) have an effect in common then
Set-of-Rules ← Set-of-Rules
Rules U find-rules(\(\{a_1\}, \{a_2\}\), Rulesize)
return Set-of-Rules

**find-rules:** (Chain1, Chain2, S)
if \(S > 0\) then
\(S ← S - 1\)
Set-of-Rules ← Set-of-Rules U \(\{\text{to-be-replaced}=\text{Chain1}, \text{replacing}=\text{Chain2}\}\)
Randomly pick a chain to expand next
say Chain1
Newchain1 ← expand(Chain1)
find-rules(Newchain1, Chain2, S)
return Set-of-Rules

**algorithm 2:** (Higher quality plan H, Lower quality plan L)
if to-be-replaced=L, replacing=H is not in the set of rules then add it
else exit
for index1 = |H| downto 1 do
for index2 = |L| downto 1 do
for \(i = 1\) to |H| do
for \(j = 1\) to |L| do
\(H ← H \cup \{h_i\}\)
\(L ← L \cup \{l_j\}\)
add to-be-replaced=L, replacing=H to Set-of-Rules

return Set-of-Rules

Figure 3: Algorithm 1: Learning rewrite rules by static domain analysis.

rewrite rule and the chain with the lower quality in the to-be-replaced part of the rewrite rule. This process of expanding a chain by adding an action continues until a user provided depth limit Rulesize is reached.

**Learning from Examples**

The problem with static learning algorithms (such as Algorithm 1) is that they learn all possible rewrite rules, many of which may never be used in any possible example. The algorithms that use the training examples to learn from do not have this problem. Such algorithms can be divided into two types: those algorithms that use the completed solutions as their input, and those algorithms that use the solution traces (records of the two solutions) as their input to learn from. Unlike the traditional learning from examples algorithms (such as EBL) whose need only one example to learn from, learning to improve quality algorithms need two example solutions to learn from: one solution of higher quality and the other of lower quality.

**By Comparing Planning Traces**

One way to learn from examples is by comparing two planning traces: the good and the bad planning trace. The good planning trace is the one that leads to a solution of higher quality and the bad planning trace is the one that leads to the solution of lower quality.

Given these two traces, the learning algorithm's first step is to retrace the bad planning-trace, looking for plan-refinement decisions that added a constraint that is not present in the good planning trace. We call such a decision point a conflicting choice point. Each conflicting choice point indicates a possible opportunity to learn. For any conflicting choice point, there are two different plan-refinement decision sequences that can be applied to a partial plan: the one added by the bad trace, and by the good trace. The application of one set of plan-refinement decisions leads to a higher quality plan and the other to a lower quality plan. However, all of the downstream planning decisions may not be relevant to resolving the flaw at the conflicting choice point. The rest of the good trace and the rest of the bad trace are then examined, with the goal of labeling a subsequent plan-refinement decision \(q\) relevant if (a) there exists a causal-link \(q \xrightarrow{c} p\) such that \(p\) is a relevant action, or (b) \(q\) binds an uninstantiated variable of a relevant open-condition.

Once both the good trace's relevant decisions and the bad trace's relevant decisions have been identified, Algorithm 2 computes (a) the actions that are added by the worse plan's relevant decision sequence. These become the action sequence to-be-replaced; (b) The actions that are added by the good trace's relevant decision sequence. These become the replacing action sequence; (c) The preconditions and effects of the replacing and the to-be-replaced action sequence. This information is then stored as a rewrite rule.

**By Comparing Two Plans**

Rewrite rules can also be learned by comparing two completed plans (a higher and a lower quality one). Figure presents an algorithm for learning plan rewriting rules by comparing two completed plans. Given two plans for the same problem, it produces all subplans of these two plans and stores them as rewrite rules with the higher quality subplan being placed in the replacing part of the rewrite rule and the lower quality subplan in the to-be-replaced part of the rule.

It is possible to make this algorithm more efficient by heuristically generating certain subplans and not all of them. One such heuristic suggested by Ambite et al. (Ambite, Knoblock, & Minton 2000) is to remove only
other idea suggested by Ambite (Ambite, Knoblock, & Minton 2000) is to run the system on various orderings of subsets of various sizes of the Set of rules and remember the smallest subset of rules that cover all the examples. However, in some situations no subset of rules may be able to cover all examples. Therefore, a more general form of this idea would be to prefer that subset that leads to the largest improvements in planning performance over the given examples. The two ideas can also be combined so that if more than one subset leads to optimal performance improvements then prefer the subset that has higher quality rewrite rules according to the goodness criteria.

Conclusion & Future Work

This paper has presented an analysis of rewrite rule learning algorithms and outlined a number of design issues involved in designing rewrite rule learning systems as well planning by rewriting systems in general. More work is needed to analyze the benefits and costs of various design choices. We are currently working on carefully designing empirical experiments to study these tradeoffs.

Acknowledgment

This work was supported by a research grant from the Natural Sciences and Engineering Research Council of Canada to the author.

References


