Plan Optimization by Plan Rewriting

José Luis Ambite, Craig A. Knoblock & Steven Minton
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292, USA
{ambite, knoblock, minton}@isi.edu

February 8, 2005

Abstract

Planning by Rewriting (PbR) is a paradigm for efficient high-quality planning that exploits declarative plan rewriting rules and efficient local search techniques to transform an easy-to-generate, but possibly suboptimal, initial plan into a high-quality plan. In addition to addressing planning efficiency and plan quality, PbR offers a new anytime planning algorithm. The plan rewriting rules can be either specified by a domain expert or automatically learned. We describe a learning approach based on comparing initial and optimal plans that produces rules competitive with manually-specified ones. PbR is fully implemented and has been applied to several existing domains. The experimental results show that the PbR approach provides significant savings in planning effort while generating high-quality plans.

1 Introduction

Planning is the process of generating a network of actions, a plan, that achieves a desired goal from an initial state of the world. Many problems of practical importance can be cast as planning problems. Instead of crafting an individual planner to solve each specific problem, a long line of research has focused on constructing domain-independent planning algorithms. Domain-independent planning accepts as input, not only descriptions of the initial state and the goal for each particular problem instance, but also a declarative domain specification, that is, the set of actions that transform a state into a new state. Domain-independent planning makes the development of planning algorithms more efficient, allows for software and domain reuse, and facilitates the principled extension of the capabilities of the planner. Unfortunately, domain-independent planning is computationally hard (Bylander, 1994; Erol, Nau, & Subrahmanian, 1995). Given the complexity limitations, most of the previous work on domain-independent planning has focused on finding *any* solution plan without careful consideration of plan quality. Usually very simple cost functions, such as the length of the plan, have been used. However, for many practical problems plan quality is crucial. In this chapter we present Planning by Rewriting (PbR), a planning paradigm that addresses both planning efficiency and plan quality while maintaining the benefits of domain independence. The framework is fully implemented and we present empirical results in several planning domains.

Two observations guided the present work. The first one is that there are two sources of complexity in planning:

- Satisfiability: the difficulty of finding any solution to the planning problem (regardless of the quality of the solution).
- Optimization: the difficulty of finding the optimal solution under a given cost metric.

For a given domain, each of these facets may contribute differently to the complexity of planning. In particular, there are many domains in which the satisfiability problem is relatively easy and their complexity is dominated by the optimization problem. For example, there may be many plans that would solve the problem, so that finding one is efficient in practice, but the cost of each solution varies greatly, thus finding the optimal one is computationally hard. We will refer to these domains as optimization domains. Some optimization domains of great practical interest are query optimization and manufacturing process planning.[1]

The second observation is that planning problems have a great deal of structure. Plans are a type of graph with strong semantics, determined by both the general properties of planning and each particular domain specification. This structure should and can be exploited to improve the efficiency of the planning process.

Prompted by the previous observations, we developed a novel approach for efficient planning in optimization domains: Planning by Rewriting (PbR). The framework works in two phases:

1. Generate an initial solution plan. Recall that in optimization domains this is efficient. However, the quality of this initial plan may be far from optimal.
2. Iteratively rewrite the current solution plan improving its quality using a set of declarative plan-rewriting rules, until either an acceptable solution is found or a resource limit is reached.

As motivation, consider the optimization domains of distributed query processing and manufacturing process planning.[2] Distributed query processing (Yu & Chang, 1984) involves generating a plan that efficiently computes a user query from data that resides at different nodes in a network. This query plan is composed of data retrieval actions at diverse information sources and operations on this data (such as those of the relational algebra: join, selection, etc). Some systems use a general-purpose planner to solve this problem (Knoblock, 1996). In this domain it is easy to construct an initial plan (any parse of the query suffices) and then transform it using a gradient-descent search to reduce its cost. The plan transformations exploit the commutative and associative properties of the (relational algebra)

[1]Interestingly, one of the most widely studied planning domains, the Blocks World, also has this property.

[2]A domain for manufacturing process planning is analyzed in detail below. The reader may want consult Figure 16 for an example of the rewriting process. The application of PbR to query planning in mediator systems is described in (Ambite & Knoblock, 2000, 2001; Ambite, 1998)

operators, and facts such as that when a group of operators can be executed together at a remote information source it is generally more efficient to do so. Figure 1 shows some sample transformations. `Simple-join-swap` transforms two join trees according to the commutative and associative properties of the join operator. `Remote-join-eval` executes a join of two subqueries at a remote source, if the source is able to do so.

Simple-Join-Swap:

$$retrieve(Q1, Source1) \bowtie [retrieve(Q2, Source2) \bowtie retrieve(Q3, Source3)] \Leftrightarrow$$
$$retrieve(Q2, Source2) \bowtie [retrieve(Q1, Source1) \bowtie retrieve(Q3, Source3)]$$

Remote-Join-Eval:

$$(retrieve(Q1, Source) \bowtie retrieve(Q2, Source)) \wedge capability(Source, join)$$
$$\Rightarrow retrieve(Q1 \bowtie Q2, Source)$$

Figure 1: Transformations in Query Planning

In manufacturing, the problem is to find an economical plan of machining operations that implement the desired features of a design. In a feature-based approach (Nau, Gupta, & Regli, 1995), it is possible to enumerate the actions involved in building a piece by analyzing its CAD model. It is more difficult to find an ordering of the operations and the setups that optimize the machining cost. However, similar to query planning, it is possible to incrementally transform a (possibly inefficient) initial plan. Often, the order of actions does not affect the design goal, only the quality of the plan, thus many actions can commute. Also, it is important to minimize the number of setups because fixing a piece on a machine is a rather time consuming operation. Interestingly, such grouping of machining operations on a setup is analogous to evaluating a subquery at a remote information source.

As suggested by these examples, there are many problems that combine the characteristics of traditional planning satisfiability with quality optimization. For these domains there often exist natural transformations that can be used to efficiently obtain high-quality plans by iterative rewriting as proposed in PbR. These transformations can be either specified by a domain expert as declarative plan-rewriting rules or learned automatically.

There are several advantages to the planning style that PbR introduces. First, PbR is a declarative domain-independent framework. This facilitates the specification of planning domains, their evolution, and the principled extension of the planner with new capabilities. Moreover, the declarative rewriting rule language provides a natural and convenient mechanism to specify complex plan transformations. Second, PbR accepts sophisticated quality measures because it operates on complete plans. Most previous planning approaches either have not addressed quality issues or have very simple quality measures, such as the number of steps in the plan, because only partial plans are available during the planning process. In general, a partial plan cannot offer enough information to evaluate a complex cost metric and/or guide the planning search effectively. Third, PbR can use local search methods that have been remarkably successful in scaling to large problems (Aarts & Lenstra, 1997). By using local search techniques, high-quality plans can be efficiently generated. Fourth, the search occurs in the space of solution plans, which is generally much smaller than the space of partial plans explored by planners based on refinement search (Kambhampati, Knoblock, &

Yang, 1995). Finally, our framework yields an anytime planning algorithm (Dean & Boddy, 1988). The planner always has a solution to offer at any point in its computation (modulo the initial plan generation that needs to be fast). This is a clear advantage over traditional planning approaches, which must run to completion before producing a solution. Thus, our system allows the possibility of trading off planning effort and plan quality. For example, in query planning the quality of a plan is its execution time and it may not make sense to keep planning if the cost of the current plan is small enough, even if a cheaper one could be found.

The remainder of the chapter is structured as follows. First, we present the basic framework of Planning by Rewriting as a domain-independent approach to local search. Second, we show experimental results comparing the basic PbR framework with other planners. Third, we present our approach to learning plan rewriting rules from examples. Fourth, we show empirically that the learned rules are competitive with manually-specified ones. Finally, we discuss related work, future work, and conclusions.

2 Planning by Rewriting as Local Search

We will describe the main issues in Planning by Rewriting as an instantiation of local search[3] (Aarts & Lenstra, 1997; Papadimitriou & Steiglitz, 1982):

- *Selection of an initial feasible point*: In PbR this phase consists of efficiently generating an initial solution plan.
- *Generation of a local neighborhood*: In PbR the neighborhood of a plan is the set of plans obtained from the application of a set of declarative plan-rewriting rules.
- *Cost function to minimize*: This is the measure of plan quality that the planner is optimizing. The plan quality function can range from a simple domain-independent cost metric, such as the number of steps, to more complex domain-specific ones, such as the query evaluation cost or the total manufacturing time for a set of parts.
- *Selection of the next point*: In PbR, this consists of deciding which solution plan to consider next. This choice determines how the global space will be explored and has a significant impact on the efficiency of planning. A variety of local search strategies can be used in PbR, such as steepest descent, simulated annealing, etc. Which search method yields the best results may be domain or problem specific.

In the following subsections we expand on these issues. First, we discuss the use of declarative rewriting rules to generate a local neighborhood of a plan. Second, we address the selection of the next plan and the associated search techniques for plan optimization. Third, we discuss the measures of plan quality. Finally, we briefly describe some approaches for initial plan generation.

[3] Although the space of rewritings can be explored by complete search methods, in the application domains we have analyzed the search space is very large and our experience suggests that local search is more appropriate. However, to what extent complete search methods are useful in a Planning by Rewriting framework remains an open issue. In this chapter we focus on local search.

2.1 Local Neighborhood Generation: Rules and Rewriting

The neighborhood of a solution plan is generated by the application of a set of declarative plan-rewriting rules. These rules embody the domain-specific knowledge about what transformations of a solution plan are likely to result in higher-quality solutions. The application of a given rule may produce one or several rewritten plans or fail to produce a plan, but the rewritten plans are guaranteed to be valid solutions. First, we describe PbR plans and the syntax and semantics of the plan-rewriting rules, both by example with a formal description. Second, we discuss two approaches to rule specification. Third, we present a taxonomy of plan-rewriting rules. Finally, we present the rewriting algorithm.

2.1.1 Plan-Rewriting Rules: Syntax and Semantics

A plan in PbR is represented by a graph, in the spirit of partial-order causal-link planners (POCL) such as UCPOP (Penberthy & Weld, 1992). In fact, PbR is implemented on top of Sage (Knoblock, 1996), which is an extension of UCPOP. Figure 2 shows a sample plan for the simple Blocks World domain of Figure 3.[4]

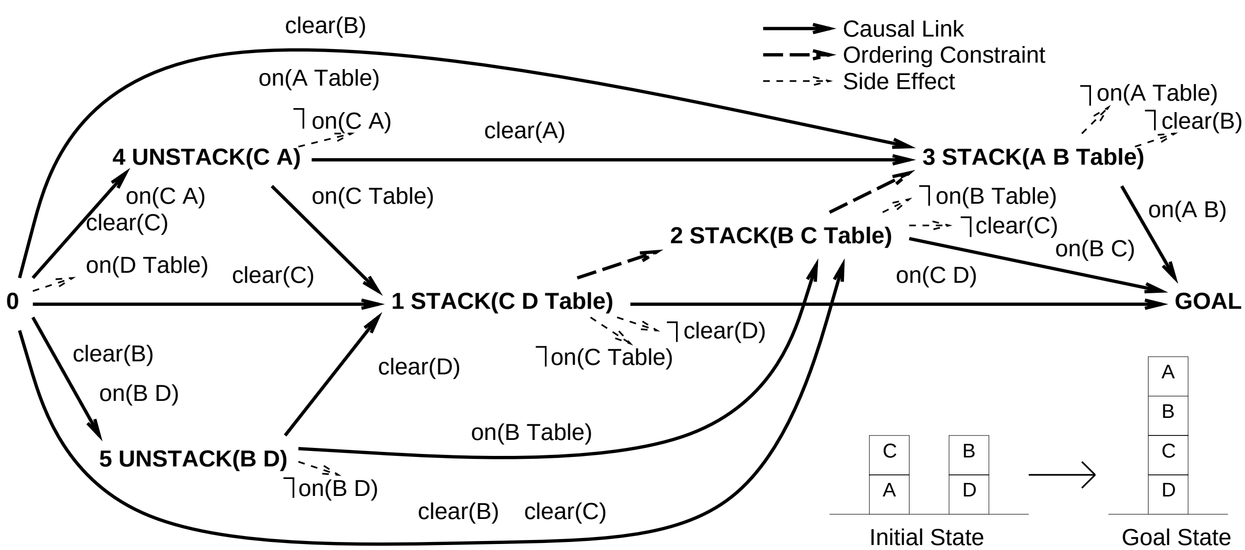


Figure 2: Sample Plan in the Blocks World Domain

A plan-rewriting rule has three components: (1) the antecedent (`:if` field) specifies a subplan to be matched; (2) the `:replace` field identifies the subplan that is going to be removed, a subset of steps and links of the antecedent; (3) the `:with` field specifies the replacement subplan. Figure 4 shows two rewriting rules for the Blocks World domain introduced in Figure 3. Intuitively, the rule `avoid-move-twice` says that, whenever possible, it is better to stack a block on top of another directly, rather than first moving it to the table. This situation occurs in plans generated by the simple algorithm that first puts all

[4] To illustrate the basic concepts in PbR, we will use examples from this simple Blocks World domain. PbR has been applied to "real-world" domains such as query planning (Ambite & Knoblock, 2001, 2000)

```
(define (operator STACK)
 :parameters (?X ?Y ?Z)
 :precondition
  (:and (on ?X ?Z) (clear ?X) (clear ?Y)
     (:neq ?Y ?Z) (:neq ?X ?Z) (:neq ?X ?Y)
     (:neq ?X Table) (:neq ?Y Table))
 :effect (:and (on ?X ?Y) (:not (on ?X ?Z))
          (clear ?Z) (:not (clear ?Y))))
```

```
(define (operator UNSTACK)
 :parameters (?X ?Y)
 :precondition
  (:and (on ?X ?Y) (clear ?X) (:neq ?X ?Y)
        (:neq ?X Table) (:neq ?Y Table))
 :effect (:and (on ?X Table) (clear ?Y)
               (:not (on ?X ?Y))))
```

Figure 3: Blocks World Operators

blocks on the table and then builds the desired towers, such as the plan in Figure 2. The rule `avoid-undo` says that the actions of moving a block to the table and back to its original position cancel each other and both actions can be removed from a plan.

```
(define-rule :name avoid-move-twice
 :if (:operators ((?n1 (unstack ?b1 ?b2))
                  (?n2 (stack ?b1 ?b3 Table)))
      :links (?n1 (on ?b1 Table) ?n2)
      :constraints ((possibly-adjacent ?n1 ?n2)
                    (:neq ?b2 ?b3)))
 :replace (:operators (?n1 ?n2))
 :with (:operators (?n3 (stack ?b1 ?b3 ?b2))))
```

```
(define-rule :name avoid-undo
 :if (:operators
       ((?n1 (unstack ?b1 ?b2))
        (?n2 (stack ?b1 ?b2 Table)))
      :constraints
       ((possibly-adjacent ?n1 ?n2))
 :replace (:operators (?n1 ?n2))
 :with NIL))
```

Figure 4: Blocks World Rewriting Rules

A rule for the manufacturing domain of (Minton, 1988) is shown in Figure 5. This domain and additional rewriting rules are described in detail in the experimental sections below. The rule states that if a plan includes two consecutive punching operations in order to make holes in two different objects, but another machine, a drill-press, is also available, the plan quality may be improved by replacing one of the punch operations with the drill-press. In this domain the plan quality is the makespan (i.e., the parallel time to manufacture all parts). This rule helps to parallelize the plan and thus improve the plan quality.

```
(define-rule :name punch-by-drill-press
  :if (:operators ((?n1 (punch ?o1 ?width1 ?orientation1))
                   (?n2 (punch ?o2 ?width2 ?orientation2)))
       :links (?n1 ?n2)
       :constraints ((:neq ?o1 ?o2)
                     (possibly-adjacent ?n1 ?n2)))
  :replace (:operators (?n1))
  :with (:operators (?n3 (drill-press ?o1 ?width1 ?orientation1))))
```

Figure 5: Manufacturing Process Planning Rewriting Rule

The plan-rewriting rule syntax follows the template shown in Figure 6. Next, we describe the semantics of the three components of a rule (`:if`, `:replace`, and `:with` fields) in detail.

The antecedent, the `:if` field, specifies a subplan to be matched against the current plan. The graph structure of the subplan is defined in the `:operators` and `:links` fields. The

```
(define-rule :name <rule-name>                         <nv> = node variable
  :if (:operators ((<nv> <np> {:resource}) ...)        <np> = node predicate
       :links ((<nv> {<lp>|:threat} <nv>) ...)         <lp> = causal link predicate
       :constraints (<ip> ...))                        <ip> = interpreted predicate
  :replace (:operators (<nv> ...)                      | = alternative
            :links ((<nv> {<lp>|:threat} <nv>) ...))   {} = optional
  :with (:operators ((<nv> <np> {:resource}) ...)
         :links ((<nv> {<lp>} <nv>) ...)))
```

Figure 6: Rewriting Rule Template

`:operators` field specifies the nodes (operators) of the graph and the `:links` field specifies the edges (causal and ordering links). Finally, the `:constraints` field specifies a set of constraints that the operators and links must satisfy.

The `:operators` field consists of a list of node variable and node predicate pairs. The step number of those steps in the plan that match the given node predicate would be correspondingly bound to the node variable. The node predicate can be interpreted in two ways: as the step action, or as a resource used by the step. For example, the node specification `(?n2 (stack ?b1 ?b3 Table))` in the antecedent of `avoid-move-twice` in Figure 4 shows a node predicate that denotes a step action. This node specification will collect tuples, composed of step number `?n2` and blocks `?b1` and `?b3`, obtained by matching steps whose action is a `stack` of a block `?b1` that is moved from the `Table` to the top of another block `?b3`. This node specification applied to the plan in Figure 2 would result in three matches: (1 C D), (2 B C), and (3 A B), for the variables `(?n2 ?b1 ?b3)` respectively. If the optional keyword `:resource` is present, the node predicate is interpreted as one of the resources used by a plan step, as opposed to describing a step action.[5] An example of a rule that matches against the resources of an operator is given in Figure 7, where the node specification `(?n1 (machine ?x) :resource)` will match all steps that use a resource of type `machine` and collect pairs of step number `?n1` and machine object `?x`.

```
(define-rule :name machine-swap
  :if (:operators ((?n1 (machine ?x) :resource)
                   (?n2 (machine ?x) :resource))
       :links ((?n1 :threat ?n2)))
  :replace (:links (?n1 ?n2))
  :with (:links (?n2 ?n1)))
```

Figure 7: Machine-Swap Rewriting Rule

The `:links` field consists of a list of link specifications. Our language admits link specifications of three types. The first type is specified as a pair of node variables. For example, `(?n1 ?n2)` in Figure 5. This specification matches any temporal ordering link in the plan, regardless if it was imposed by a causal link or by the resolution of a threat.

The second type of link specification matches causal links. Causal links are specified as triples composed of the node variable of the producer step, an link predicate, and the

[5] In Sage and PbR, resources are associated to operators, see (Knoblock, 1996) for details.

node variable of the consumer step. The semantics of a causal link is that the producer step asserts in its effects the predicate, which in turn is needed in the preconditions of the consumer step. For example, the link specification `(?n1 (on ?b1 Table) ?n2)` in Figure 4 matches steps `?n1` that put a block `?b1` on the `Table` and steps `?n2` that subsequently pick up this block. That link specification applied to the plan in Figure 2 would result in the matches: (4 C 1) and (5 B 2), for the variables `(?n1 ?b1 ?n2)`.

The third type of link specification matches ordering links originating from the resolution of threats (coming either from resource conflicts or from operator conflicts). These links are selected by using the keyword `:threat` in the place of a condition. For example, the `machine-swap` rule in Figure 7 uses the link specification `(?n1 :threat ?n2)` to ensure that only steps that are ordered because they are involved in a threat situation are matched. This helps to identify which are the "critical" steps that do not have any other reasons (i.e. causal links) to be in such order, and therefore this rule may attempt to reorder them. This is useful when the plan quality depends on the degree of parallelism in the plan as a different ordering may help to parallelize the plan. Recall that threats can be solved either by promotion or demotion, so the reverse ordering may also produce a valid plan, which is often the case when the conflict is among resources as in the rule in Figure 7.

Interpreted predicates, built-in and user-defined, can be specified in the `:constraints` field. These predicates are implemented programmatically as opposed to being obtained by matching against components from the plan. The built-in predicates currently implemented are inequality(`:neq`), comparison (`< <= > >=`), and arithmetic (`+ - * /`) predicates. The user can also add arbitrary predicates and their corresponding programmatic implementations. The interpreted predicates may act as filters on the previous variables or introduce new variables (and compute new values for them). For example, the user-defined predicate `possibly-adjacent` in the rules in Figure 4 ensures that the steps are consecutive in some linearization of the plan.[6] For the plan in Figure 2 the extension of the `possibly-adjacent` predicate is: (0 4), (0 5), (4 5), (5 4), (4 1), (5 1), (1 2), (2 3), and (3 Goal).

The user can easily add interpreted predicates by including a function definition that implements the predicate. During rule matching our algorithm passes arguments and calls such functions when appropriate. The current plan is passed as a default first argument to the interpreted predicates in order to provide a context for the computation of the predicate (but it can be ignored). Figure 8 show a skeleton for the (Lisp) implementation of the `possibly-adjacent` and `less-than` interpreted predicates.

```
(defun possibly-adjacent (plan node1 node2)
  (not (necessarily-not-adjacent
        node1
        node2
        ;; accesses the current plan
        (plan-ordering plan))))
```

```
(defun less-than (plan n1 n2)
  (declare (ignore plan))
  (when (and (numberp n1) (numberp n2))
    (if (< n1 n2)
        '(nil) ;; true
      nil))) ;; false
```

Figure 8: Sample Implementation of Interpreted Predicates

[6]The interpreted predicate `possibly-adjacent` makes the link expression in the antecedent of the `avoid--move-twice` rule in Figure 4 redundant. Unstack puts the block `?b1` on the table from where it is picked up by the stack operator, thus the causal link `(?n1 (on ?b1 Table) ?n2)` is already implied.

The consequent is composed of the `:replace` and `:with` fields. The `:replace` field specifies the subplan that is going to be removed from the plan, which is a subset of the steps and links identified in the antecedent. If a step is removed, all the links that refer to the step are also removed. The `:with` field specifies the replacement subplan. As we will see later, the replacement subplan does not need to be completely specified. For example, the `:with` field of the `avoid-move-twice` rule of Figure 4 only specifies the addition of a `stack` step but not how this step is embedded into the plan. The links to the rest of the plan are automatically computed during the rewriting process.

2.1.2 Plan-Rewriting Rules: Full versus Partial Specification

PbR gives the user total flexibility in defining rewriting rules. In this section we describe two approaches to guaranteeing that a rewriting rule specification preserves plan correctness, that is, produces a valid rewritten plan when applied to a valid plan.

In the *full-specification* approach the rule specifies *all* steps and links involved in a rewriting. The rule antecedent identifies all the anchoring points for the operators in the consequent, so that the embedding of the replacement subplan is unambiguous and results in a valid plan. The burden of proving the rule correct lies upon the user or an automated rule defining procedure. These kind of rules are the ones typically used in graph rewriting systems (Schürr, 1997).

In the *partial-specification* approach the rule defines the operators and links that constitute the gist of the plan transformation, but the rule does not prescribe the precise embedding of the replacement subplan. The burden of producing a valid plan lies upon the system. PbR takes advantage of the semantics of domain-independent planning to accept such a relaxed rule specification, fill in the details, and produce a valid rewritten plan. Moreover, the user is free to specify rules that may not necessarily be able to compute a rewriting for a plan that matches the antecedent because some necessary condition was not checked in the antecedent. That is, a partially-specified rule may be overgeneral. This may seem undesirable, but often a rule may cover more useful cases and be more naturally specified in this form. The rule may only fail for rarely occurring plans, so that the effort in defining and matching the complete specification may not be worthwhile. In any case, the plan-rewriting algorithm ensures that the application of a rewriting rule either generates a valid plan or fails to produce a plan (Theorem 1 in (Ambite & Knoblock, 2001)).

As an example of these two approaches to rule specification, consider the `avoid-move--twice-full` rule of Figure 9, which is a fully-specified version of the `avoid-move-twice` rule of Figure 4. The `avoid-move-twice-full` rule is more complex and less natural to specify than `avoid-move-twice`. But, more importantly, `avoid-move-twice-full` is making more commitments than `avoid-move-twice`. In particular, `avoid-move-twice-full` fixes the producer of `(clear ?b1)` for `?n3` to be `?n4` when `?n7` is also known to be a valid candidate. In general, there are several alternative producers for a precondition of the replacement subplan, and consequently many possible embeddings. A different fully-specified rule is needed to capture each embedding. The number of rules grows exponentially as all permutations of the embeddings are enumerated. However, by using the partial-specification approach we can express a general plan transformation by a single natural rule.

```
(define-rule :name avoid-move-twice-full
  :if (:operators ((?n1 (unstack ?b1 ?b2))
                   (?n2 (stack ?b1 ?b3 Table)))
       :links ((?n4 (clear ?b1) ?n1) (?n5 (on ?b1 ?b2) ?n1)
               (?n1 (clear ?b2) ?n6) (?n1 (on ?b1 Table) ?n2)
               (?n7 (clear ?b1) ?n2) (?n8 (clear ?b3) ?n2)
               (?n2 (on ?b1 ?b3) ?n9))
       :constraints ((possibly-adjacent ?n1 ?n2)
                     (:neq ?b2 ?b3)))
  :replace (:operators (?n1 ?n2))
  :with (:operators ((?n3 (stack ?b1 ?b3 ?b2)))
         :links ((?n4 (clear ?b1) ?n3) (?n8 (clear ?b3) ?n3)
                 (?n5 (on ?b1 ?b2) ?n3) (?n3 (on ?b1 ?b3) ?n9))))
```

Figure 9: Fully-specified Rewriting Rule

In summary, the main advantage of the full-specification rules is that the rewriting can be performed more efficiently because the embedding of the consequent is already specified. The disadvantages are that the number of rules to represent a generic plan transformation may be very large and the resulting rules quite lengthy; both of these problems may decrease the performance of the match algorithm. Also, the rule specification is error prone if written by the user. Conversely, the main advantage of the partial-specification rules is that a single rule can represent a complex plan transformation naturally and concisely. The rule can cover a large number of plan structures even if it may occasionally fail. Also, the partial specification rules are much easier to specify and understand by the users of the system. As we have seen, PbR provides a high degree of flexibility for defining plan-rewriting rules.

2.1.3 A Taxonomy of Plan-Rewriting Rules

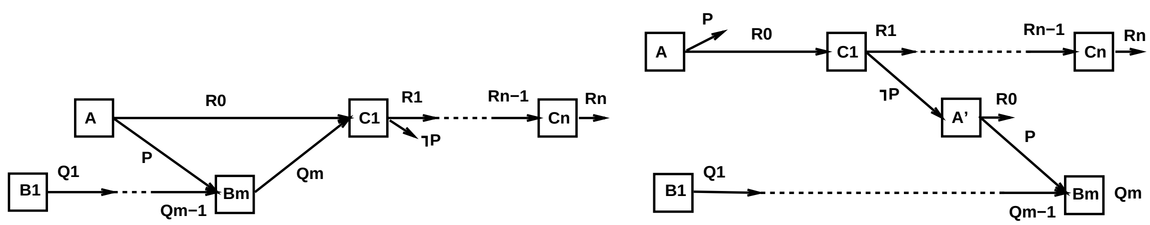
In order to guide the user in defining plan-rewriting rules for a domain or to help in designing algorithms to automatically deduce the rules from the domain specification, it is helpful to know what kinds of rules are useful. We have identified the following general types of transformation rules:

Reorder: These are rules based on algebraic properties of the operators, such as commutative, associative and distributive laws. For example, the commutative rule that reorders two operators that need the same resource in Figure 7, or the `simple-join-swap` rule in Figure 1 that combines the commutative and associative properties of the relational algebra.

Collapse: These are rules that replace a subplan by a smaller subplan. For example, when several operators can be replaced by one, as in the `remote-join-eval` rule in Figure 1. This rule replaces two remote retrievals at the same information source and a local join by a single remote join operation (when the remote source has the capability of performing joins). Other examples are the Blocks World rules in Figure 4 that replace an `unstack` and a `stack` operators either by an equivalent single `stack` operator or the empty plan.

Expand: These are rules that replace a subplan by a bigger subplan. Although this may appear counter-intuitive initially, it is easy to imagine a situation in which an expensive operator can be replaced by a set of operators that are cheaper as a whole. An interesting case is when some of these operators are already present in the plan and can be synergistically reused. We did not find this rule type in the domains analyzed so far, but Bäckström (1994) presents a framework in which adding actions improves the quality of the plans. His quality metric is the plan execution time, similarly to the manufacturing domain of our experiments below. Figure 10 shows an example of a planning domain where adding actions improves quality (from Bäckström, 1994). In this example, removing the link between `Bm` and `C1` and inserting a new action `A'` shortens significantly the time to execute the plan.

Parallelize: These are rules that replace a subplan with an equivalent alternative subplan that requires fewer ordering constraints. A typical case is when there are redundant or alternative resources that the operators can use. For example, the rule `punch-by-drill-press` in Figure 5. Another example is the rule that Figure 10 suggests that could be seen as a combination of the expand and parallelize types.



(a) Low Quality Plan (b) High Quality Plan

Figure 10: Adding Actions Can Improve Quality

2.1.4 Plan-Rewriting Algorithm

The plan-rewriting algorithm is shown in Figure 11. The algorithm takes two inputs: a valid plan P, and a rewriting rule $R = (q_m,\ p_r,\ p_c)$, where q_m is the antecedent query, p_r is the replaced subplan, and p_c is the replacement subplan. The output is a valid rewritten plan P'. To disentangle the algorithm from any particular search strategy, we write it using non-deterministic choice as is customary.

The matching of the antecedent of the rewriting rule (q_m) determines if the rule is applicable and identifies the steps and links of interest (line 1). This matching can be seen as subgraph isomorphism between the antecedent subplan and the current plan (with the results then filtered by applying the `:constraints`). However, we take a different approach. PbR implements rule matching as conjunctive query evaluation. Our implementation keeps a relational representation of the steps and links in the current plan similar to the node and link specifications of the rewriting rules. For example, the database for the plan in Figure 2 contains one table for the `unstack` steps with schema `(?n1 ?b1 ?b2)` and tuples (4 C A) and (5 B D), another table for the causal links involving the `clear` condition with schema `(?n1 ?n2 ?b)` and tuples (0 1 C), (0 2 B), (0 2 C), (0 3 B), (0 4 C), (0 5 B), (4 3 A) and (5 1 D), and similar tables for the other operator and link types. The match process consists of interpreting the rule antecedent as a conjunctive query with interpreted predicates, and executing this query against the relational view of the plan structures. As a running example, we will analyze the application of the `avoid-move-twice` rule of Figure 4 to the plan in

Figure 2. Matching the rule antecedent identifies steps 1 and 4. More precisely, considering the antecedent as a query, the result is the single tuple (4 C A 1 D) for the variables `(?n1 ?b1 ?b2 ?n2 ?b3)`.

---

procedure *RewritePlan*
Input: a valid partial-order plan P
a rewriting rule $R = (q_m, p_r, p_c)$, $Variables(p_r) \subseteq Variables(q_m)$
Output: a valid rewritten partial-order plan P' (or failure)

1. $\Sigma := Match(q_m, P)$
   Match the rule antecedent q_m (`:if` field) against P. The result is a set of substitutions $\Sigma = \{..., \sigma_i, ...\}$ for variables in q_m.
2. **If** $\Sigma = \emptyset$ then **return** failure
3. **Choose** a match $\sigma_i \in \Sigma$
4. $p_r^i := \sigma_i p_r$
   Instantiate the subplan to be removed p_r (the `:replace` field) according to σ_i.
5. $P_r^i := AddFlaws(UsefulEffects(p_r^i), P - p_r^i)$
   Remove the instantiated subplan p_r^i from the plan P and add the UsefulEffects of p_r^i as open conditions. The resulting plan P_r^i is now incomplete.
6. $p_c^i := \sigma_i p_c$
   Instantiate the replacement subplan p_c (the `:with` field) according to σ_i.
7. $P_c^i := AddFlaws(Preconditions(p_c^i) \cup FindThreats(P_r^i \cup p_c^i), P_r^i \cup p_c^i)$
   Add the instantiated replacement subplan p_c^i to P_r^i. Find new threats and open conditions and add them as flaws. P_c^i is potentially incomplete, having several flaws that need to be resolved.
8. **Choose** $P' \in rPOP(P_c^i)$
   Complete the plan using a partial-order causal-link planning algorithm (restricted to do only step reuse, but no step addition) in order to resolve threats and open conditions. $rPOP$ returns failure if no valid plan can be found.
   Non-deterministically choose a completion.
9. **Return** P'

Figure 11: Plan-Rewriting Algorithm

After (non-deterministically) choosing a match σ_i to work on (line 3), the algorithm instantiates the subplan specified by the `:replace` field (p_r) according to such match (line 4) and removes the instantiated subplan p_r^i from the original plan P (line 5). All the edges incoming and emanating from nodes of the replaced subplan are also removed. The effects that the replaced plan p_r^i was achieving for the remainder of the plan ($P - p_r^i$), the *UsefulEffects* of p_r^i, will now have to be achieved by the replacement subplan (or other steps of $P - p_r^i$). In order to facilitate this process, the *AddFlaws* procedure records these effects as open conditions.[7] The result is the partial plan P_r^i (line 5). Continuing with our example,

[7] POCL planners operate by keeping track and repairing *flaws* found in a partial plan. Open conditions, operator threats, and resource threats are collectively called flaws (Penberthy & Weld, 1992). *AddFlaws(F,P)* adds the set of flaws F to the plan structure P.

Figure 12(a) shows the plan resulting from removing steps 1 and 4 from the plan in Figure 2.

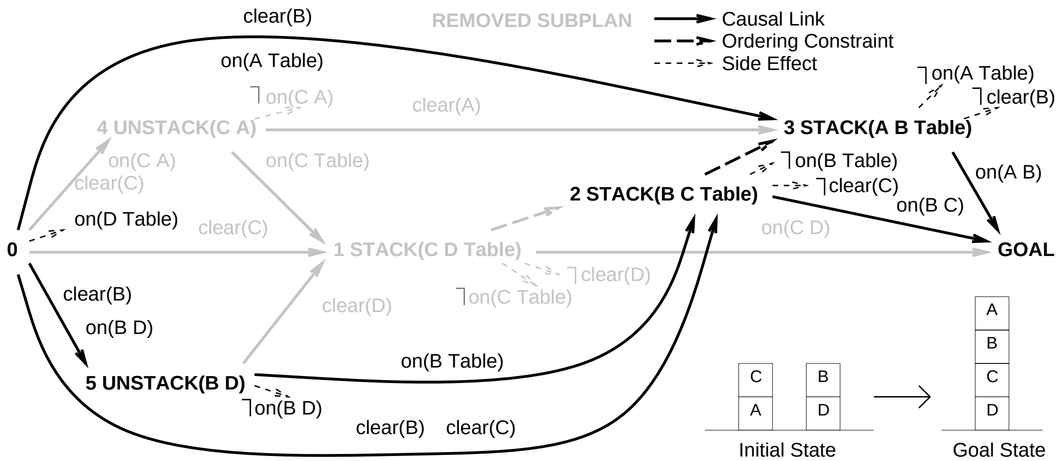
Finally, the algorithm embeds the instantiated replacement subplan p_c^i into the remainder of the original plan (lines 6-9). If the rule is completely specified, the algorithm simply adds the (already instantiated) replacement subplan to the plan, and no further work is necessary. If the rule is partially specified, the algorithm computes the embeddings of the replacement subplan into the remainder of the original plan in three stages. First, the algorithm adds the instantiated steps and links of the replacement plan p_c^i (line 6) into the current partial plan P_r^i (line 7). Figure 12(b) shows the state of our example after p_c^i, the new `stack` step (`6`), has been incorporated into the plan. Note the open conditions `(clear A)` and `on(C D)`. Second, the *FindThreats* procedure computes the possible threats, both operator threats and resource conflicts, occurring in the $P_r^i \cup p_c^i$ partial plan (line 7); for example, the threat situation on the `clear(C)` proposition between step 6 and 2 in Figure 12(b). These threats and the preconditions of the replacement plan p_c^i are recorded by *AddFlaws* resulting in the partial plan P_c^i. Finally, the algorithm completes the plan using *rPOP*, a partial-order causal-link planning procedure restricted to only reuse steps (i.e., no step addition) (line 8). *rPOP* allows us to support our expressive operator language and to have the flexibility for computing one or all embeddings. If only one rewriting is needed, *rPOP* stops at the first valid plan. Otherwise, it continues until exhausting all alternative ways of satisfying open preconditions and resolving conflicts, which produces all valid rewritings. In our running example, only one embedding is possible and the resulting plan is that of Figure 12(c), where the new `stack` step (`6`) produces `(clear A)` and `on(C D)`, its preconditions are satisfied, and the ordering `(6 2)` ensures that the plan is valid.

In (Ambite & Knoblock, 2001) we show that the plan rewriting algorithm of Figure 11 is sound, in the sense that it produces a valid plan if the input is a valid plan, or it outputs failure if the input plan cannot be rewritten using the given rule. Since each elementary plan-rewriting step is sound, the sequence of rewritings performed during PbR's optimization search is also sound.

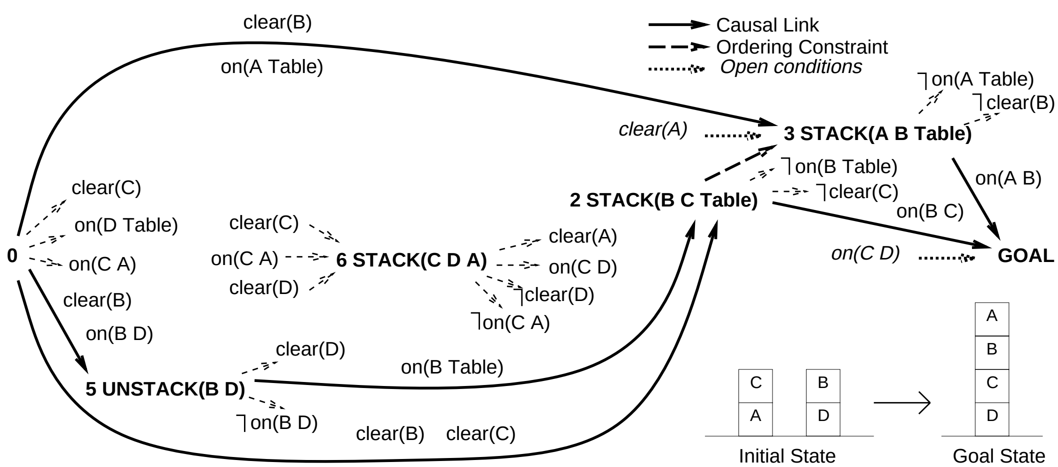
We cannot guarantee that PbR's optimization search is complete in the sense that the optimal plan would be found. PbR uses local search and it is well known that, in general, local search cannot be complete. Even if PbR exhaustively explores the space of plan rewritings induced by a given initial plan and a set of rewriting rules, we still cannot prove that all solution plans will be reached. This is a property of the initial plan generator, the set of rewriting rules, and the semantics of the planning domain. The rewriting rules of PbR play a similar role as traditional declarative search control where the completeness of the search may be traded for efficiency. An open problem is whether using techniques for inferring invariants in a planning domain (Gerevini & Schubert, 1998; Fox & Long, 1998; Rintanen, 2000) and/or proving convergence of term and graph rewriting systems (Baader & Nipkow, 1998), could provide conditions for completeness of a plan-rewriting search in a given planning domain.

2.2 Selection of Next Plan: Search Strategies

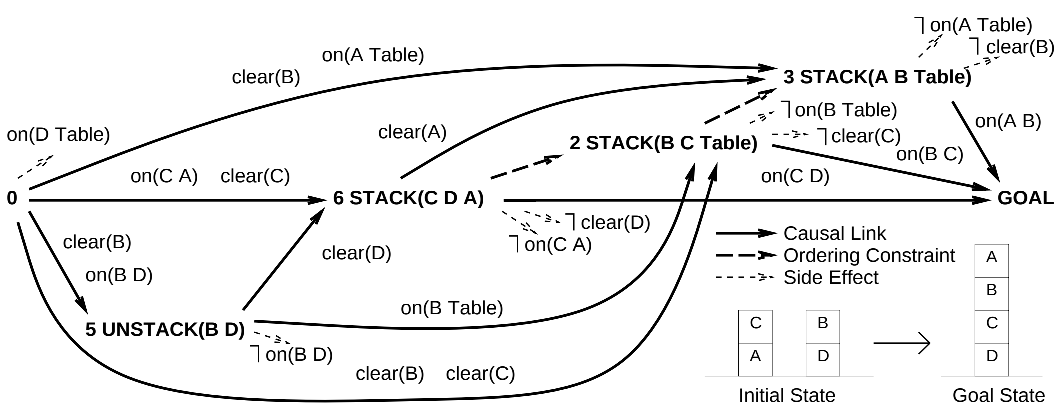
Many local search methods, such as first and best improvement, simulated annealing (Kirkpatrick, Gelatt, & Vecchi, 1983), tabu search (Glover, 1989), or variable-depth search (Lin & Kernighan, 1973), can be applied straightforwardly to PbR. Figure 13 depicts graphically



(a) Application of a Rewriting Rule: After Removing Subplan



(b) Application of a Rewriting Rule: After Adding Replacement Subplan



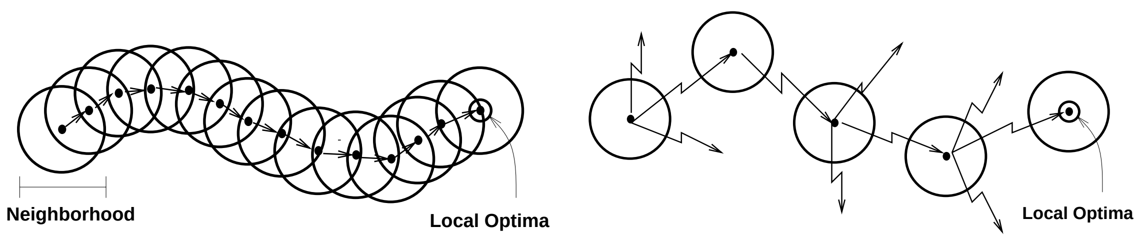
(c) Rewritten Plan

Figure 12: Plan Rewriting: Applying rule `avoid-move-twice` of Figure 4 to plan of Figure 2

the behavior of iterative improvement and variable-depth search. In our experiments below we have used first and best improvement, which have performed well. Next, we describe some details of the application of these two methods in PbR.

First improvement generates the rewritings incrementally and selects the first plan of better cost than the current one. In order to implement this method efficiently we can use a tuple-at-a-time evaluation of the rule antecedent, similarly to the behavior of Prolog. Then, for that rule instantiation, generate one embedding, test the cost of the resulting plan, and if it is not better that the current plan, repeat. We have the choice of generating another embedding of the same rule instantiation, generate another instantiation of the same rule, or generate a match for a different rule.

Best improvement generates the complete set of rewritten plans and selects the best. This method requires computing all matches and all embeddings for each match. All the matches can be obtained by evaluating the rule antecedent as a set-at-a-time database query. In our experience, computing the plan embeddings was usually more expensive than computing the rule matches.



(a) Basic Iterative Improvement (b) Variable-Depth Search

Figure 13: Local Search

2.3 Plan Quality

In most practical planning domains the quality of the plans is crucial. This is one of the motivations for the Planning by Rewriting approach. In PbR the user defines the measure of plan quality most appropriate for the application domain. This quality metric could range from a simple domain-independent cost metric, such as the number of steps, to more complex domain-specific ones. For example, in query planning the measure of plan quality usually is an estimation of the query execution cost based on the size of the database relations, the data manipulation operations involved in answering a query, and the cost of network transfer. In (Ambite & Knoblock, 2000) we describe a complex cost metric, based on traditional query estimation techniques (Silberschatz, Korth, & Sudarshan, 1997), that PbR uses to optimize query plans. The cost metric may involve actual monetary costs if some of the information sources require payments. In the job-shop scheduling domain some simple cost functions are the makespan, or the sum of the times to finish each piece. A more sophisticated manufacturing domain may include a variety of concerns such as the cost, reliability, and precision of each operator/process, the costs of resources and materials used by the operators, the utilization of the machines, etc.

A significant advantage of PbR is that the complete plan is available to assess its quality. In generative planners the complete plan is not available until the search for a solution is completed, so usually only very simple plan quality metrics, such as the number of steps, can be used. Moreover, if the plan cost is not additive, a plan refinement strategy is impractical since it may need to exhaustively explore the search space to find the optimal plan. An example of non-additive cost function appears in the UNIX planning domain (Etzioni & Weld, 1994) where a plan to transfer files between two machines may be cheaper if the files are compressed initially (and uncompressed after arrival). That is, the plan that includes the compression (and the necessary uncompression) operations is more cost effective, but a plan refinement search would not naturally lead to it. By using complete plans, PbR can accurately assess arbitrary measures of quality.

2.4 Initial Plan Generation

PbR relies on an efficient plan generator to produce the initial plan on which to start the optimization process. Fortunately, the efficiency of planners has increased significantly in recent years. Much of these gains come from exploiting heuristics or domain-dependent search control. However, the quality of the generated plans is often far from optimal, thus the need for an optimization process like PbR. We briefly review some approaches to efficiently generate initial plans.

HSP (Bonet & Geffner, 2001) applies heuristic search to classical AI planning. The domain-independent heuristic function is a relaxed version of the planning problem: it computes the number of required steps to reach the goal disregarding negated effects in the operators. Such metric can be computed efficiently. Despite its simplicity and that the heuristic is not admissible, it scales surprisingly well for many domains. Because the plans are generated according to the fixed heuristic function, the planner cannot incorporate a quality metric.

TLPlan (Bacchus & Kabanza, 2000) is an efficient forward-chaining planner that uses domain-dependent search control expressed in temporal logic. Because in forward chaining the complete state is available, much more refined domain control knowledge can be specified. The preferred search strategy used by TLPlan is depth-first search, so although it finds plans efficiently, the plans may be of low quality. Note that because it is a generative planner that explores partial sequences of steps, it cannot use sophisticated quality measures.

Some systems automatically learn search control for a given planning domain or even specific problem instances. Minton (1988) shows how to deduce search control rules by applying explanation-based learning to problem-solving traces. Another approach to automatically generating search control is by analyzing statically the operators (Etzioni, 1993) or inferring invariants in the planning domain (Gerevini & Schubert, 1998; Fox & Long, 1998; Rintanen, 2000). Abstraction provides yet another form of search control. Knoblock (1994) presents a system that automatically learns abstraction hierarchies from a planning domain or a particular problem instance in order to speed up planning.

By setting the type of search and providing a strong bias by means of the search control rules, the planner can quickly generate a valid, although possibly suboptimal, initial plan. For example, in the manufacturing domain of (Minton, 1988), analyzed in detail in the experimental section, depth-first search and a goal selection heuristic based on abstraction hierarchies (Knoblock, 1994) quickly generates a feasible plan, but often the quality of this plan, which is defined as the time required to manufacture all objects, is suboptimal.

3 Empirical Results

In this section we show the broad applicability of Planning by Rewriting by analyzing three domains with different characteristics: a process manufacturing domain (Minton, 1988), a transportation logistics domain, and the Blocks World domain that we used in the examples throughout the chapter. An analysis of a domain for query planning in data integration systems appears in (Ambite & Knoblock, 2000, 2001; Ambite, 1998).

3.1 Manufacturing Process Planning

The task in the manufacturing process planning domain is to find a plan to manufacture a set of parts. We implemented a PbR translation of the domain specification of (Minton, 1988). This domain contains a variety of machines, such as a lathe, punch, spray painter, welder, etc, for a total of ten machining operations. Some of the operators of the specification appear in Figure 14 (see (Ambite & Knoblock, 2001; Ambite, 1998) for the full description).

The measure of plan cost is the makespan (or schedule length), the (parallel) time to manufacture *all* parts. In this domain all of the machining operations are assumed to take unit time. The machines and the objects (parts) are modeled as resources in order to enforce that only one part can be placed on a machine at a time and that a machine can only operate on a single part at a time (except `bolt` and `weld` which operate on two parts simultaneously).

We have already shown some of the types of rewriting rules for this domain in Figures 5 and 7. Figure 15 shows some additional rules that we used in our experiments. Rules `IP-by-SP` and `roll-by-lathe` exchange operators that are equivalent with respect to achieving some effects. By examining the operator definitions in Figure 14, it can be readily noticed that both `immersion-paint` and `spray-paint` change the value of the `painted` predicate. Similarly, `roll-by-lathe` exchanges `roll` and `lathe` operators as they both make parts cylindrical. To focus the search on the most promising exchanges these rules only match operators in the critical path (by means of the interpreted predicate `in-critical-path`).

The two bottom rules in Figure 15 are more sophisticated. The `lathe+SP-by-SP` rule takes care of an undesirable effect of the simple depth-first search used by our initial plan generator. In this domain, in order to spray paint a part, the part must have a regular shape. Being cylindrical is a regular shape, therefore the initial planner may decide to make the part cylindrical by lathing it in order to paint it! However, this may not be necessary as the part may already have a regular shape (for example, it could be rectangular, which is also a regular shape). Thus, the `lathe+SP-by-SP` substitutes the pair `spray-paint` and `lathe` by a single `spray-paint` operation. The supporting `regular-shapes` interpreted predicate just enumerates which are the regular shapes. These rules are partially specified and are not guaranteed to always produce a rewriting. Nevertheless, they are often successful in producing plans of lower cost.

```
(define (operator LATHE)
 :parameters (?x)
 :resources ((machine LATHE) (is-object ?x))
 :precondition (is-object ?x)
 :effect
  (:and (:forall (?color)
          (:not (painted ?x ?color)))
        (:forall (?shape)
          (:when (:neq ?shape CYLINDRICAL)
            (:not (shape ?x ?shape))))
        (:forall (?surf)
          (:when (:neq ?surf ROUGH)
            (:not (surface-condition ?x ?surf))))
        (surface-condition ?x ROUGH)
        (shape ?x CYLINDRICAL)))

(define (operator DRILL-PRESS)
 :parameters (?x ?width ?orientation)
 :resources ((machine DRILL-PRESS)
             (is-object ?x))
 :precondition
   (:and (is-object ?x)
         (have-bit ?width)
         (is-drillable ?x ?orientation))
 :effect (has-hole ?x ?width ?orientation))

(define (operator IMMERSION-PAINT)
 :parameters (?x ?color)
 :resources ((machine IMMERSION-PAINTER)
             (is-object ?x))
 :precondition
   (:and (is-object ?x)
         (have-paint-for-immersion ?color))
 :effect (painted ?x ?color))

(define (operator WELD)
 :parameters (?x ?y ?new-obj ?orient)
 :resources ((machine WELDER)
             (is-object ?x) (is-object ?y))
 :precondition
   (:and (is-object ?x) (is-object ?y)
         (composite-object ?new-obj ?orient ?x ?y)
         (can-be-welded ?x ?y ?orient))
 :effect (:and (temperature ?new-obj HOT)
               (joined ?x ?y ?orient)
               (:not (is-object ?x))
               (:not (is-object ?y))))
```

```
(define (operator ROLL)
 :parameters (?x)
 :resources ((machine ROLLER) (is-object ?x))
 :precondition (is-object ?x)
 :effect
  (:and (:forall (?color)
          (:not (painted ?x ?color)))
        (:forall (?shape)
          (:when (:neq ?shape CYLINDRICAL)
            (:not (shape ?x ?shape))))
        (:forall (?temp)
          (:when (:neq ?temp HOT)
            (:not (temperature ?x ?temp))))
        (:forall (?surf)
          (:not (surface-condition ?x ?surf)))
        (:forall (?width ?orientation)
          (:not (has-hole ?x ?width ?orientation)))
        (temperature ?x HOT)
        (shape ?x CYLINDRICAL)))
(define (operator PUNCH)
 :parameters (?x ?width ?orientation)
 :resources ((machine PUNCH) (is-object ?x))
 :precondition
  (:and (is-object ?x)
        (has-clamp PUNCH)
        (is-punchable ?x ?width ?orientation))
 :effect
  (:and (:forall (?surf)
          (:when (:neq ?surf ROUGH)
            (:not (surface-condition ?x ?surf))))
        (surface-condition ?x ROUGH)
        (has-hole ?x ?width ?orientation)))
(define (operator SPRAY-PAINT)
 :parameters (?x ?color ?shape)
 :resources ((machine SPRAY-PAINTER)
             (is-object ?x))
 :precondition (:and (is-object ?x)
                     (sprayable ?color)
                     (temperature ?x COLD)
                     (regular-shape ?shape)
                     (shape ?x ?shape)
                     (has-clamp SPRAY-PAINTER))
 :effect (painted ?x ?color))
(define (operator BOLT)
 :parameters (?x ?y ?new-obj ?orient ?width)
 :resources ((machine BOLTER)
             (is-object ?x) (is-object ?y))
 :precondition
  (:and (is-object ?x) (is-object ?y)
        (composite-object ?new-obj ?orient ?x ?y)
        (has-hole ?x ?width ?orient)
        (has-hole ?y ?width ?orient)
        (bolt-width ?width)
        (can-be-bolted ?x ?y ?orient))
 :effect (:and (:not (is-object ?x))
               (:not (is-object ?y))
               (joined ?x ?y ?orient)))
```

Figure 14: (Some) Operators for Manufacturing Process Planning

The `both-providers-diff-bolt` rule is an example of rules that explore bolting two parts using bolts of different size if fewer operations may be needed for the plan. We developed these rules by analyzing differences in the quality of the optimal plans and the rewritten plans. This rule states that if the parts to be bolted already have compatible holes in them, it is better to reuse those operators that produced the holes. The initial plan generator may have drilled (or punched) holes whose only purpose was to bolt the parts. However, the goal of the problem may already require some holes to be performed on the parts to be joined. Reusing the available holes produces a more economical plan.

```
(define-rule :name IP-by-SP
  :if (:operators (?n1 (immersion-paint ?x ?c))
       :constraints ((regular-shapes ?s)
                     (in-critical-path ?n1)))
  :replace (:operators (?n1))
  :with (:operators (?n2 (spray-paint ?x ?c ?s))))
(define-rule :name lathe+SP-by-SP
  :if (:operators
       ((?n1 (lathe ?x))
        (?n2 (spray-paint ?x ?color ?shape1)))
       :constraints ((regular-shapes ?shape2)))
  :replace (:operators (?n1 ?n2))
  :with (:operators
          ((?n3 (spray-paint ?x ?color ?shape2)))))

(define-rule :name roll-by-lathe
  :if (:operators ((?n1 (roll ?x)))
       :constraints ((in-critical-path ?n1)))
  :replace (:operators (?n1))
  :with (:operators (?n2 (lathe ?x))))

(define-rule :name both-providers-diff-bolt
  :if (:operators ((?n3 (bolt ?x ?y ?z ?o ?w1)))
       :links ((?n1 (has-hole ?x ?w1 ?o) ?n3)
               (?n2 (has-hole ?y ?w1 ?o) ?n3)
               (?n4 (has-hole ?x ?w2 ?o) ?n5)
               (?n6 (has-hole ?y ?w2 ?o) ?n7))
       :constraints ((:neq ?w1 ?w2)))
  :replace (:operators (?n1 ?n2 ?n3))
  :with (:operators ((?n8 (bolt ?x ?y ?z ?o ?w2)))
         :links ((?n4 (has-hole ?x ?w2 ?o) ?n8)
                 (?n6 (has-hole ?y ?w2 ?o) ?n8))))
```

Figure 15: Rewriting Rules for Manufacturing Process Planning

As an illustration of the rewriting process in the manufacturing domain, consider Figure 16. The plan at the top of the figure is the result of a simple initial plan generator that solves each part independently and concatenates the corresponding subplans. Although such plan is generated efficiently, it is of poor quality. It requires six time-steps to manufacture all parts. The figure shows the application of two rewriting rules, `machine-swap` and `IP-by-SP`, that improve the quality of this plan. The operators matched by the rule antecedent are shown in *italics*. The operators introduced in the rule consequent are shown in **bold**. First, the `machine-swap` rule reorders the punching operations on parts `A` and `B`. This breaks the long critical path that resulted from the simple concatenation of their respective subplans. The schedule length improves from six to four time-steps. Still, the three parts `A`, `B`, and `C` use the same painting operation (`immersion-paint`). As the immersion-painter can only process one piece at a time, the three operations must be done serially. Fortunately, in our domain there is another painting operation: `spray-paint`. The `IP-by-SP` rule takes advantage of this fact and substitutes an `immersion-paint` operation on part `B` by a `spray-paint` operation. This further parallelizes the plan obtaining a schedule length of three time-steps, which is the optimal for this plan.

We compare four planners (IPP, Initial, and two configurations of PbR):

IPP: This is one of the most efficient domain-independent planners (Koehler, Nebel, Hoffman, & Dimopoulos, 1997) of the planning competition held at the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98). IPP (Koehler et al., 1997) is an optimized re-implementation and extension of Graphplan (Blum & Furst, 1997). IPP

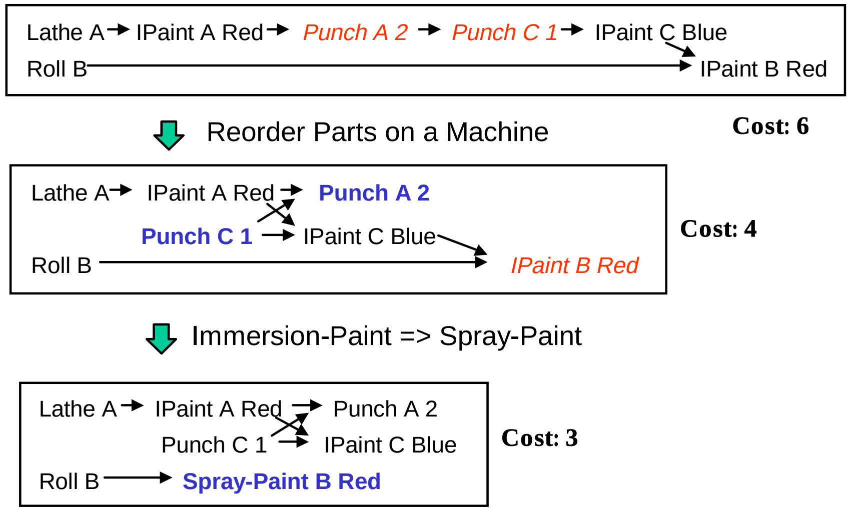


Figure 16: Rewriting in the Manufacturing Domain

produces shortest parallel plans. For our manufacturing domain, this is exactly the schedule length, the cost function that we are optimizing.[8]

Initial: The initial plan generator uses a divide-and-conquer heuristic in order to generate plans as fast as possible. First, it produces subplans for each part and for the `joined` goals independently. These subplans are generated by Sage using a depth-first search without any regard to plan cost. Then, it concatenates the subsequences of actions and merges them into a POCL plan.

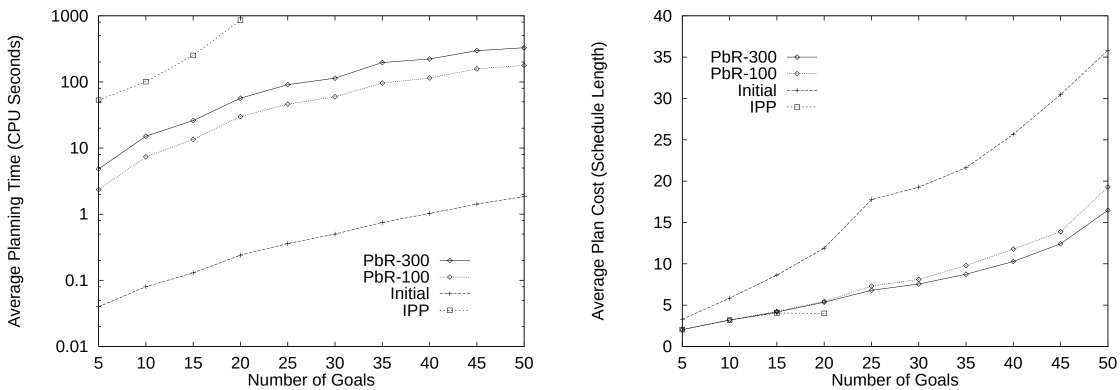
PbR: We present results for two configurations of PbR, which we will refer to as PbR-100 and PbR-300. Both configurations use a first improvement gradient search strategy with random walk on the cost plateaus. The rewriting rules used are those of Figure 15. For each problem PbR starts its search from the plan generated by Initial. The two configurations differ only on how many total plateau plans are allowed. PbR-100 allows considering up to 100 plans that do not improve the cost without terminating the search. Similarly, PbR-300 allows 300 plateau plans. Note that the limit is across all plateaus encountered during the search for a problem, not for each plateau.

We tested each of the four systems on 200 problems, for machining 10 parts, ranging from 5 to 50 goals. The goals are distributed randomly over the 10 parts. So, for the 50-goal problems, there is an average of 5 goals per part. The results are shown in Figure 17. In these graphs each data point is the average of 20 problems for each given number of goals. There were 10 provably unsolvable problems. Initial and PbR solved all 200 problems (or proved them unsolvable). IPP solved 65 problems in total: all problems at 5 and 10 goals, 19 at 15 goals, and 6 at 20 goals. IPP could not solve any problem with more than 20 goals under the 1000 CPU seconds time limit.

Figure 17(a) shows the average time on the solvable problems for each problem set for

[8] Although IPP is a domain-independent planner, we compare it to PbR to test whether the additional knowledge provided by the plan rewriting rules is useful both in planning time and in plan quality.

the four planners. Figure 17(b) shows the average schedule length for the problems solved by *each* of the planners for the 50 goal range. The fastest planner is Initial, but it produces plans with a cost of about twice the optimal. IPP produces the optimal plans, but it cannot solve problems of more than 20 goals. The PbR configurations scale gracefully with the number of goals, improving considerably the cost of the plans generated by Initial. The additional exploration of PbR-300 allows it to improve the plans even further. For the range of problems solved by IPP, PbR-300 matches the optimal cost of the IPP plans (except in one problem) and the faster PbR-100 also stays very close to the optimal (less than 2.5% average cost difference).[9]



(a) Average Planning Time (b) Average Plan Cost

Figure 17: Performance: Manufacturing Process Planning

3.2 Logistics

The task in the logistics domain is to transport several packages from their initial location to their desired destinations. We used a version of the `logistics-strips` planning domain of the AIPS98 planning competition which we restricted to using only trucks but not planes.[10] The domain is shown in Figure 18. A package is transported from one location to another by loading it into a truck, driving the truck to the destination, and unloading the truck. A truck can load any number of packages. The cost function is the (parallel) time to deliver all packages (measured as the number of operators in the critical path of a plan).

We compare three planners on this domain:

IPP: IPP produces optimal plans in this domain.

Initial: The initial plan generator picks a distinguished location and delivers packages one by one starting and returning to the distinguished location. For example, assume that truck t1 is at the distinguished location l1, and package p1 must be delivered from location

[9]The reason for the difference between PbR and IPP at the 20-goal complexity level is because the cost results for IPP are only for the 6 problems that it could solve, while the results for PbR and Initial are the average of all of the 20 problems, PbR matches the cost of these 6 optimal plans produced by IPP

[10]In the logistics domain of AIPS98, the problems of moving packages by plane among different cities and by truck among different locations in a city are isomorphic, so we focused on only one of them to better analyze how the rewriting rules can be learned (Ambite, Knoblock, & Minton, 2000).

```
(define (operator LOAD-TRUCK)
 :parameters (?obj ?truck ?loc)
 :precondition
  (:and (obj ?obj) (truck ?truck) (location ?loc)
        (at ?truck ?loc) (at ?obj ?loc))
 :effect (:and (:not (at ?obj ?loc))
               (in ?obj ?truck)))

(define (operator UNLOAD-TRUCK)
 :parameters (?obj ?truck ?loc)
 :precondition
  (:and (obj ?obj) (truck ?truck) (location ?loc)
        (at ?truck ?loc) (in ?obj ?truck))
 :effect (:and (:not (in ?obj ?truck))
               (at ?obj ?loc)))

(define (operator DRIVE-TRUCK)
 :parameters (?truck ?loc-from ?loc-to ?city)
 :precondition (:and (truck ?truck) (location ?loc-from) (location ?loc-to) (city ?city)
                     (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?city))
 :effect (:and (:not (at ?truck ?loc-from)) (at ?truck ?loc-to)))
```

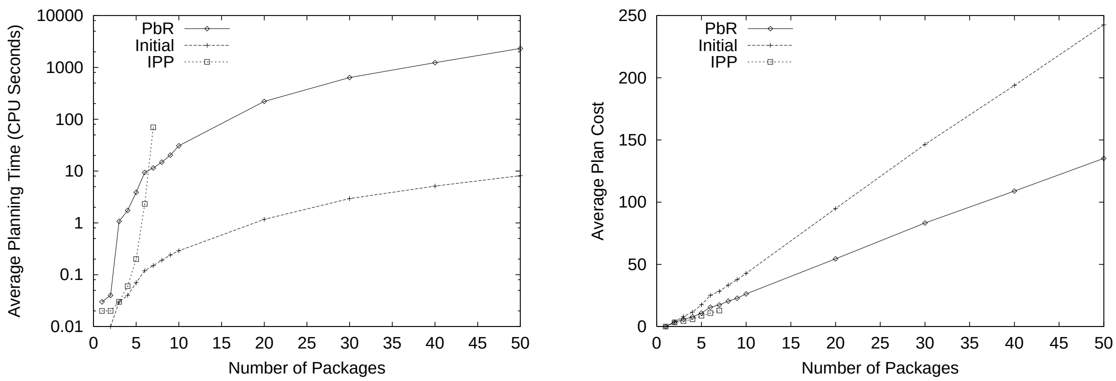
Figure 18: Operators for Logistics

l2 to location l3. The plan would be: `drive-truck(t1 l1 l2 c)`, `load-truck(p1 t1 l2)`, `drive-truck(t1 l2 l3 c)`, `unload-truck(p1 t1 l3)`, `drive-truck(t1 l3 l1 c)`. The initial plan generator would keep producing these circular trips for the remaining packages. Although this algorithm is very efficient it produces plans of very low quality.

PbR: PbR starts from the plan produced by Initial and uses the plan rewriting rules shown in Figure 19 to optimize plan quality. The `loop` rule states that driving to a location and returning back immediately after is useless. The fact that the operators must be adjacent is important because it implies that no intervening `load` or `unload` was performed. In the same vein, the `triangle` rule states that it is better to drive directly between two locations than through a third point if no other operation is performed at such point. The `load-earlier` rule captures the situation in which a package is not loaded in the truck the first time that the package's location is visited. This occurs when the initial planner was concerned with a trip for another package. The `unload-later` rule captures the dual case. PbR applies a first improvement search strategy with only one run (no restarts).

```
(define-rule :name loop
  :if (:operators
       ((?n1 (drive-truck ?t ?l1 ?l2 ?c))
        (?n2 (drive-truck ?t ?l2 ?l1 ?c)))
      :links ((?n1 ?n2))
      :constraints
       ((adjacent-in-critical-path ?n1 ?n2)))
  :replace (:operators (?n1 ?n2))
  :with NIL)

(define-rule :name triangle
  :if (:operators
       ((?n1 (drive-truck ?t ?l1 ?l2 ?c))
        (?n2 (drive-truck ?t ?l2 ?l3 ?c)))
      :links ((?n1 ?n2))
      :constraints
       ((adjacent-in-critical-path ?n1 ?n2)))
  :replace (:operators (?n1 ?n2))
  :with (:operators
         ((?n3 (drive-truck ?t ?l1 ?l3 ?c)))))

(define-rule :name load-earlier
  :if (:operators
       ((?n1 (drive-truck ?t ?l1 ?l2 ?c))
        (?n2 (drive-truck ?t ?l3 ?l2 ?c))
        (?n3 (load-truck ?p ?t ?l2)))
      :links ((?n2 ?n3))
      :constraints
       ((adjacent-in-critical-path ?n2 ?n3)
        (before ?n1 ?n2)))
  :replace (:operators (?n3))
  :with (:operators ((?n4 (load-truck ?p ?t ?l2)))
         :links ((?n1 ?n4))))

(define-rule :name unload-later
  :if (:operators
       ((?n1 (drive-truck ?t ?l1 ?l2 ?c))
        (?n2 (unload-truck ?p ?t ?l2))
        (?n3 (drive-truck ?t ?l3 ?l2 ?c)))
      :links ((?n1 ?n2))
      :constraints
       ((adjacent-in-critical-path ?n1 ?n2)
        (before ?n2 ?n3)))
  :replace (:operators (?n2))
  :with (:operators ((?n4 (unload-truck ?p ?t ?l2)))
         :links ((?n3 ?n4))))
```

Figure 19: Logistics Rewriting Rules



(a) Average Planning Time (b) Average Plan Cost

Figure 20: Performance: Logistics

We compared the performance of IPP, Initial, and PbR on a set of logistics problems involving up to 50 packages. Each problem instance has the same number of packages, locations, and goals. There was a single truck and a single city. The performance results are shown in Figure 20. In these graphs each data point is the average of 20 problems for each given number of packages. All the problems were satisfiable. IPP could only solve problems up to 7 packages (it also solved 10 out of 20 for 8 packages, and 1 out of 20 for 9 packages, but these are not shown in the figure). Figure 20(a) shows the average planning time. Figure 20(b) shows the average cost for the 50 packages range. The results are similar to the previous experiment. Initial is efficient but highly suboptimal. PbR is able to considerably improve the cost of these plans and approach the optimal.

3.3 Blocks World

We implemented a classical Blocks World domain with the two operators in Figure 3. This domain has two actions: `stack` that puts one block on top of another, and, `unstack` that places a block on the table to start a new tower. Plan quality in this domain is simply the number of steps. Optimal planning in this domain is NP-hard (Gupta & Nau, 1992). However, it is trivial to generate a correct, but suboptimal, plan in linear time using the naive algorithm: put all blocks on the table and build the desired towers from the bottom up. We compare three planners on this domain:

IPP: In this experiment we used the GAM goal ordering heuristic (Koehler & Hoffmann, 2000) that had been tested in Blocks World problems with good scaling results.

Initial: This planner is a programmatic implementation of the naive linear-time algorithm. This algorithm produces plans of length no worse than twice the optimal.

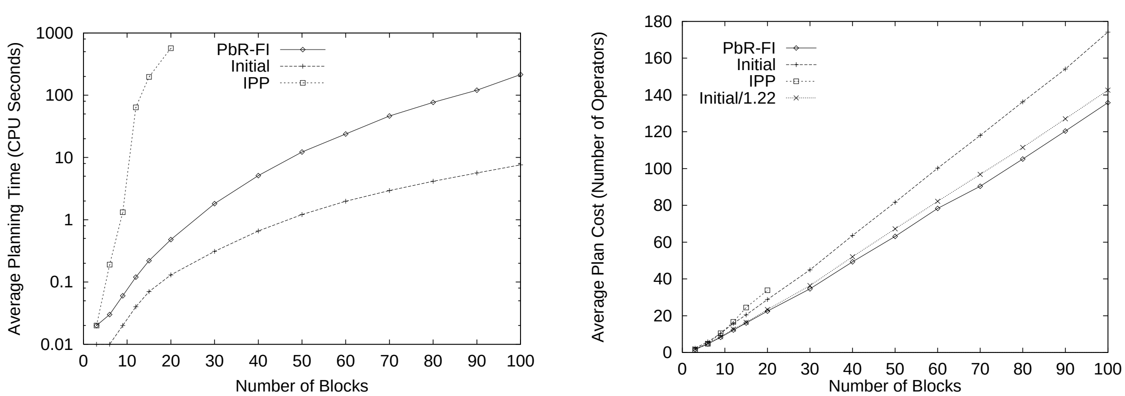
PbR: This configuration of PbR starts from the plan produced by Initial and uses the two plan-rewriting rules shown in Figure 4 to optimize plan quality. PbR applies a first improvement strategy with only one run (no restarts).

We generated random Blocks World problems scaling the number of blocks. The problem set consists of 350 random problems (25 problems at each of the 3, 6, 9, 12, 15, 20, 30, 40, 50, 60, 70, 80, 90, and 100 blocks level). The problems may have multiple towers in the initial state and in the goal state.

Figure 21(a) shows the average planning time of the 25 problems for each block quantity. IPP cannot solve problems with more than 20 blocks within the time limit of 1000 CPU seconds. The local search of PbR allows it to scale much better and solve all the problems.

Figure 21(b) shows the average plan cost as the number of blocks increases. PbR improves considerably the quality of the initial plans. The optimal quality is only known for very small problems, where PbR approximates it, but does not achieve it (we ran Sage for problems of less than 9 blocks). For larger plans we do not know the optimal cost. However, Slaney & Thiébaux (1996) performed an extensive experimental analysis of Blocks World planning using a domain like ours. In their comparison among different approximation algorithms they found that our initial plan generator (unstack-stack) achieves empirically a quality around 1.22 the optimal for the range of problem sizes we have analyzed (Figure 7 in Slaney & Thiébaux, 1996). The value of our average initial plans divided by 1.22 suggests the quality of the optimal plans. The quality achieved by PbR is comparable with that value. In fact it is slightly better which may be due to the relatively small number of problems tested (25 per block size) or to skew in our random problem generator. Interestingly the plans found by IPP are actually of low quality. This is due to the fact that IPP produces shortest parallel plans. That means that the plans can be constructed in the fewest time steps, but IPP may introduce more actions in each time step than are required.

In summary, the experiments in this and the previous sections show that across a variety of domains PbR scales to large problems while still producing high-quality plans.



(a) Average Planning Time (b) Average Plan Cost

Figure 21: Performance: Blocks World

4 Learning Plan Rewriting Rules

Despite the advantages of PbR in terms of scalability, plan quality, and anytime behavior, the framework we have described so far requires more inputs from the designer than other planning approaches. In addition to the operator specification, initial state, and goal that domain-independent planners take as input, PbR also requires an initial plan generator, a set of plan rewriting rules, and a search strategy (see Figure 22(a)). Although the plan rewriting rules can be conveniently specified in a high-level declarative language, designing and selecting which rules are the most appropriate requires a thorough understanding of the properties of the planning domain and requires the most effort by the designer.

In this section we address this limitation by providing a method for learning the rewriting rules from examples. The main idea is to solve a set of training problems for the planning domain using both the initial plan generator and an optimal planner. Then, the system compares the initial and optimal plan and hypothesizes a rewriting rule that would transform one into the other. An schematic of the resulting system is shown in Figure 22(b). Some ideas on automating the other inputs are discussed in the future work section.

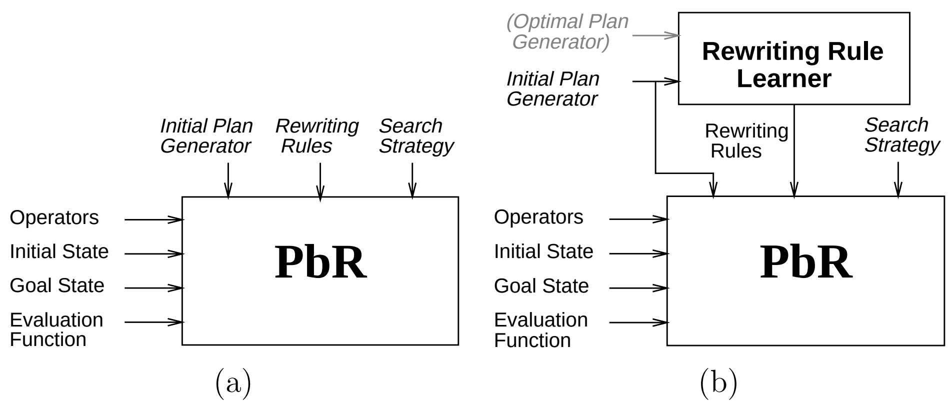


Figure 22: Basic PbR (a) and PbR with Rewriting Rule Learning (b)

4.1 Rule Generation

The main assumption of our learning algorithm is that useful rewriting rules are of relatively small size (measured as the number of nodes and edges in the rule). If a domain requires large rewriting rules, it is probably not a good candidate for a local search, iterative repair algorithm such as PbR. Previous research also lends support for biases that favor conciseness (Minton & Underwood, 1994). The rule generation algorithm follows these steps:

1. Problem Generation. To start the process, our algorithm needs a set of training problems for the planning domain. The choice of training problems determines the rules learned. Ideally, we would like problems drawn from the target problem distribution that generate plans gradually increasing in size (i.e., number of plan steps) in order to learn the smallest rewriting rules first. Towards this end we have explored two heuristics, based on a random problem generator, that work well in practice. For some domains the size of the plans can be controlled accurately by the number of goals. Thus, our system generates sets of problems increasing the number of goals up to a given goal size. For each goal size the system generates a number of random problems. We used this heuristic in our experiments.

An alternative strategy is to generate a large number of problems with different goal sizes, sort the resulting plans by increasing size, and select the first N to be the training set.

2. Initial Plan Generation. For each domain, we define an initial plan generator as described earlier. For example, the plan of Figure 2 which was generated by putting all blocks on the table and building the desired towers from the bottom up.

3. Optimal Plan Generation. Our algorithm uses a general purpose planner performing a complete search according to the given cost metric to find the optimal plan. This is feasible only because the training problems are small; otherwise, the search space of the complete planner would explode. In our implementation we have used IPP and Sage as the optimal planners. For example, Figure 12(c) shows the optimal plan for the problem in Figure 2.

4. Plan Comparison. Both the initial and optimal plans are ground labeled graphs. Our algorithm performs graph differences between the initial and the optimal plans to identify nodes and edges present in only one of the plans. Formally, an intersection graph G_i of two graphs G_1 and G_2 is a maximal subgraph isomorphism between G_1 and G_2. If in a graph there are nodes with identical labels, there may be several intersection graphs. Given a graph intersection G_i, a graph difference $G_1 - G_2$ is the subgraph of G_1 whose nodes and edges are not in G_i. In the example of Figures 2 and 12(c), the graph difference between the initial and the optimal plans, $G_{ini} - G_{opt}$, is the graph formed by the nodes: `unstack(C A)` and `stack(C D Table)`; and the edges: `(0 clear(C) 1)`, `(0 clear(C) 4)`, `(0 on(C A) 4)`, `(1 on(C D) Goal)`, `(4 clear(A) 3)`, `(4 on(C Table) 1)`, `(5 clear(D) 1)`, and `(1 2)`. Similarly, $G_{opt} - G_{ini}$ is formed by the nodes: `stack(C D A)`, and the edges: `(6 clear(A) 3)`, `(5 clear(D) 6)`, `(0 clear(C) 6)`, `(0 on(C A) 6)`, `(6 on(C D) Goal)`, and `(6 2)`.

5. Ground Rule Generation. After the plan comparison, the nodes and edges present only in the initial plan form the basis for the antecedent of the rule, and those present only in the optimal plan form the basis for the consequent. In order to maximize the applicability of the rule, not all the differences in nodes and edges of the respective graphs are included. Specifically, if there are nodes in the difference, only the edges internal to those nodes are included in the rule. This amounts to removing from consideration the edges that link the nodes to the rest of the plan. In other words, we are generating partially-specified rules. In our example, the antecedent nodes are `unstack(C A)` (node 4) and `stack(C D Table)` (node `1`). Therefore, the only internal edge is `(4 on(C Table) 1)`. This edge is included in the rule antecedent and the other edges are ignored. As the consequent is composed of only one node, there are no internal edges. Rule `bw-1-ground` in Figure 23 is the ground rule proposed from the plans of Figures 2 and 12(c).

If there are only edge (ordering or causal link) differences between the antecedent and the consequent, a rule including only edge specifications may be overly general. To provide some context for the application of the rule our algorithm includes in the antecedent specification those nodes participating in the differing edges (see rule `sc-14` in Figure 27 for an example).

6. Rule Generalization. Our algorithm generalizes the ground rule conservatively by replacing constants by variables, except when the schemas of the operators logically imply a constant in some position of a predicate (similarly to EBL (Minton, 1988)). Rule `bw-1-generalized` in Figure 23 is the generalization of rule `bw-1-ground` which was learned from the plans of Figures 2 and 12(c). The constant `Table` remains in the `bw-1-generalized` rule as is it imposed by the effects of `unstack` (see Figure 3).

```
(define-rule :name bw-1-ground
 :if (:operators ((?n1 (unstack C A))
                  (?n2 (stack C D Table)))
     :links (?n1 (on C Table) ?n2))
 :replace (:operators (?n1 ?n2))
 :with (:operators (?n3 (stack C D A))))
```

```
(define-rule :name bw-1-generalized
 :if (:operators ((?n1 (unstack ?b1 ?b2))
                  (?n2 (stack ?b1 ?b3 Table)))
     :links (?n1 (on ?b1 Table) ?n2))
 :replace (:operators (?n1 ?n2))
 :with (:operators (?n3 (stack ?b1 ?b3 ?b2))))
```

Figure 23: Ground vs. Generalized Rewriting Rules

4.2 Biasing towards Small Rules

There may be a large number of differences between an initial and an optimal plan. These differences are often better understood and explained as a sequence of small rewritings than as the application of a large monolithic rewriting. Therefore, in order to converge to a set of small "primitive" rewriting rules, our system applies the algorithm in Figure 24.

The main ideas behind the algorithm are to identify the smallest rule first and to simplify the current plans before learning additional rules. First, the algorithm generates initial and optimal plans for a set of sample problems. Then, it enters a loop that brings the initial plans increasingly closer to the optimal plans. The crucial steps are `6` and `3`. In step `6` the smallest rewriting rule (`r`) is chosen first.[11] This rule is applied to each of the current plans. If it improves the quality of some plan, the rule enters the set of learned rules (`L`). Otherwise, the algorithm tries the next smallest rule in the current generation. Step `3` applies all previously learned rules to the current initial plans in order to simplify the plans as much as possible before starting a new generation of rule learning. This helps in generating new rules that are small and that do not subsume a previously learned rule. The algorithm terminates when no more cost-improving rules can be found.

5 Empirical Results for PbR using Learned Rules

We tested our learning algorithm on the same three domains described before: the Blocks World domain used along the chapter, the manufacturing process planning domain of (Minton, 1988), and our restricted logistics domain.

[11] The size of a rule is the number of the conditions in both antecedent and consequent. Ties are broken in favor of the rule with the smallest consequent

```
Converge-to-Small-Rules:
1.  Generate a set of sample problems (P),
      initial plans (I), and optimal plans (O).
2.  Initialize the current plans (C)
      to the initial plans (I).
3.  Apply previously learned rules (L) to C
      (L is initially empty).
4.  Propose rewriting rules (R) for pairs of
      current initial (C) and optimal (O) plans
      (if their cost differ).
5.  If no cost-improving rules, Then Go to 10.
6.  S := Order the rules (R) by size.
7.  Extract the smallest rule (r) from S.
8.  Apply rule r to each current initial plan (in C)
      repeatedly until the rule does not produce
      any quality improvement.
9.  If r produced an improvement in some plan,
       Then Add r to the set of learned rules (L)
            The rewritten plans form the new C.
            Go to 3.
       Else Go to 7.
10. Return L
```

Figure 24: Bias towards small rules

5.1 Blocks World

Our learning algorithm proposed the three rules shown in Figure 25, based on 15 random problems involving 3, 4, and 5 goals (5 problems each). Figure 4 shows the two manually-defined plan rewriting rules for this domain. Rules `bw-1` and `bw-2` in Figure 25 are essentially the same as rules `avoid-move-twice` and `avoid-undo` in Figure 4, respectively. The main difference is the interpreted predicate `possibly-adjacent` that acts as a filter to improve the efficiency of the manual rules, but is not critical to the rule efficacy. The authors thought that the manual rules in Figure 4 were sufficient for all practical purposes, but our learning algorithm discovered an additional rule (`bw-3`) that addresses an optimization not covered by the two manual rules. Sometimes the blocks are in the desired position in the initial state, but our initial plan generator unstacks all blocks regardless. Rule `bw-3` would remove such unnecessary `unstack` operators. Note that our rewriting engine always produces valid plans. Therefore, if a plan cannot remain valid after removing a given `unstack`, this rule will not produce a rewriting.

We compared the performance of the manual and learned rules on the Blocks World as the number of blocks increases. We tested four planners: `Initial`, `IPP` (with the GAM heuristic); `PbR-Manual`, PbR with the manually-specified rules of Figure 4; and `PbR-Learned`, PbR with the learned rules of Figure 25.

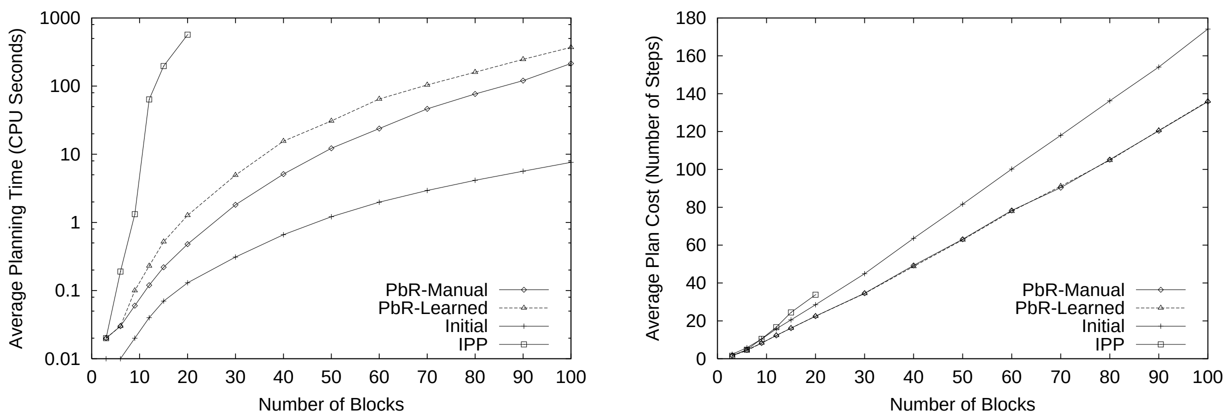
Figure 26(a) shows the average planning time of the 25 problems for each block quantity. IPP cannot solve problems with more than 20 blocks within a time limit of 1000 CPU seconds. Both configurations of PbR scale much better than IPP, solving all the problems. Empirically, the manual rules were more efficient than the learned rules by a constant factor.

```
(define-rule :name bw-1 ;; avoid-move-twice-learned
  :if (:operators ((?n1 (unstack ?b1 ?b2))
                   (?n2 (stack ?b1 ?b3 Table)))
       :links ((?n1 (on ?b1 Table) ?n2)))
  :replace (:operators (?n1 ?n2))
  :with (:operators (?n3 (stack ?b1 ?b3 ?b2))))

(define-rule :name bw-2 ;; avoid-undo-learned
  :if (:operators ((?n1 (unstack ?b1 ?b2))
                   (?n2 (stack ?b1 ?b2 Table)))
       :links ((?n1 (on ?b1 Table) ?n2)
               (?n1 (clear ?b2) ?n2)))
  :replace (:operators ((?n1 ?n2)))
  :with nil)

(define-rule :name bw-3 ;; useless-unstack-learned
  :if (:operators ((?n1 (unstack ?b1 ?b2))))
  :replace (:operators ((?n1)))
  :with nil)
```

Figure 25: Learned Rewriting Rules: Blocks World

The reason is that there are two manual rules versus three learned ones, and that the manual rules benefit from an additional filtering condition as we discussed above.

Figure 26(b) shows the average plan cost as the number of blocks increases. PbR improves considerably the quality of the initial plans. The optimal quality is only known for very small problems, where PbR approximates it.[12] The learned rules match the quality of the manual rules (the lines for PbR overlap in Figure 26(b)). Moreover, in some problems the learned rules actually produce lower cost plans due to the additional rule (`bw-3`) that removes unnecessary `unstack` operators.



(a) Planning Time (b) Plan Cost

Figure 26: Performance with Learned Rules: Blocks World

[12]We ran Sage for the 3 and 6-block problems. We used IPP for the purpose of comparing planning time. However, IPP optimizes a different cost metric, shortest parallel time-steps, instead of number of plan steps.

5.2 Manufacturing Process Planning

We ran our learning algorithm on 200 random problems involving 2, 3, 4, and 5 goals (50 problems each) on ten objects. The system learned a total of 18 rewriting rules, including some of the most interesting manual rules defined earlier. For example, the rule `lathe+SP-by-SP`, shown in Figure 15, was manually specified after a careful analysis of the depth-first search used by the initial plan generator. Our learning algorithm discovered the corresponding rule `sc-8` (Figure 27). The learned rule does not use the `regular-shapes` interpreted predicate (which enumerates the regular shapes), but it is just as general because the free variable `?shape2` in the rule consequent will capture any valid constant.

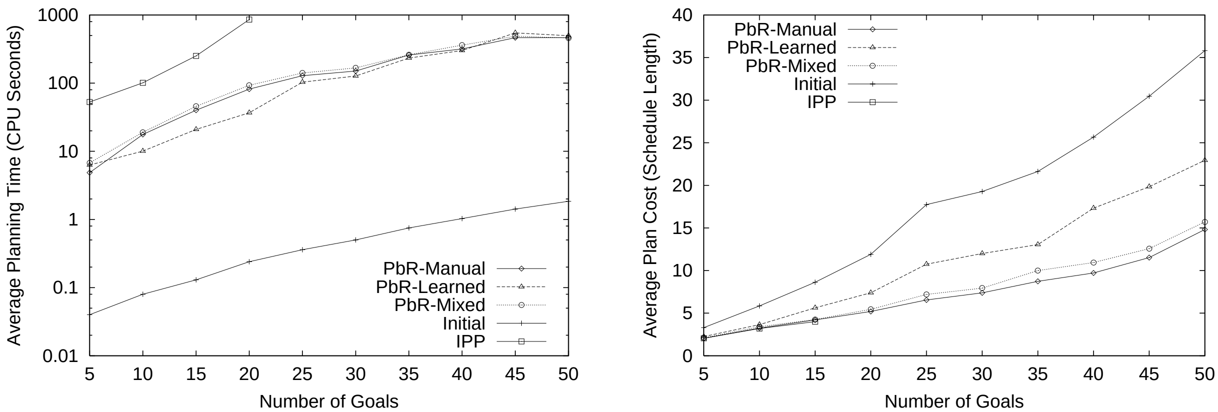
The rules `machine-swap` in Figure 7 and `sc-14` in Figure 27 show a limitation of our current learning algorithm, namely, that it does not learn over the resource specifications in the operators. The manually-defined `machine-swap` rule allows the system to explore the possible orderings of operations that require the same machine. This rule finds two consecutive operations on the same machine and swaps their order. Our learning system produced more specific rules that are versions of this principle, but it did not capture all possible combinations. Rule `sc-14` is one such learned rule. This rule would be subsumed by the `machine-swap`, because the `punch` is a machine resource. This is not a major limitation of our framework and we plan to extend the basic rule generation mechanism to also learn over resource specifications.

```
(define-rule :name sc-8            ;; Learned
  :if (:operators
       ((?n1 (lathe ?x))
        (?n2 (spray-paint ?x ?color Cylindrical))))
  :replace (:operators (?n1 ?n2))
  :with (:operators
         (?n3 (spray-paint ?x ?color ?shape2))))

(define-rule :name sc-14           ;; Learned
  :if (:operators ((?n1 (punch ?x ?w1 ?o))
                   (?n2 (punch ?y ?w1 ?o)))
       :links ((?n1 ?n2)))
  :replace (:links ((?n1 ?n2)))
  :with (:links ((?n2 ?n1))))
```

Figure 27: (Some) Learned Rewriting Rules: Manufacturing

We compared the performance of the manual and learned rules for the manufacturing process planning domain with the same experimental setting as before. We tested five planners: `Initial`; `IPP`, which produces the optimal plans; `PbR-Manual`, PbR with the manually-specified rules in Figure 15; `PbR-Learned`, PbR with the learned rules; and `PbR-Mixed`, which adds to the learned rules the two manually-specified rules that deal with resources (the `machine-swap` rule in Figure 27, and a similar one on objects).

The results are shown in Figure 28. In these graphs each data point is the average of 20 problems for each given number of goals. There were 10 provably unsolvable problems. Initial, and thus PbR, solved all the 200 problems (or proved them unsolvable). IPP only solved 65 problems under the 1000 CPU seconds time limit: all problems at 5 and 10 goals, 19 at 15 goals, and 6 at 20 goals. Figure 28(a) shows the average planning time on the solvable problems. Figure 28(b) shows the average schedule length for the problems solved by the planners for the 50 goal range. The fastest planner is Initial, but it produces plans with a cost of more that twice the optimal (which is produced by IPP). The three configurations of PbR scale much better than IPP solving all problems. The manual rules achieve a quality very close to the optimal (where optimal cost is known, and scale gracefully thereafter). The learned rules improve significantly the quality of the initial plans, but they do not reach the optimal quality because many of the resource swap rules are missing. Finally, when we add the two general resource-swap rules to the the learned rules (`PbR-Mixed`), the cost achieved approaches that of the manual rules.



(a) Average Planning Time (b) Average Plan Cost

Figure 28: Performance with Learned Rules: Manufacturing

5.3 Logistics

Our system learned the rules in Figure 29 from a set of 60 problems with 2, 4, and 5 goals (20 problems each). Rules `logs-1` and `logs-3` capture the same transformations as rules `loop` and `triangle`, respectively. Rule `logs-2` chooses a different starting point for a trip. Rule `logs-3` is the most interesting of the learned rules as it was surprisingly effective in optimizing the plans. Rule `logs-3` seems to be an overgeneralization of rule `triangle`, but precisely by not requiring that the nodes are `adjacent-in-critical-path`, it applies in a greater number of situations.

We compared the performance of the manual and learned rules on a set of logistics problems involving up to 50 packages. Each problem instance has the same number of packages, locations, and goals. There was a single truck and a single city. We tested four planners: `Initial`, the sequential circular-trip initial plan generator described above; `IPP`, which produces optimal plans; `PbR-Manual`, PbR with the manually-specified rules in Figure 19; and `PbR-Learned`, PbR with the learned rules of Figure 29.

The performance results are shown in Figure 30. In these graphs each data point is the average of 20 problems for each given number of packages. All the problems were satisfiable.

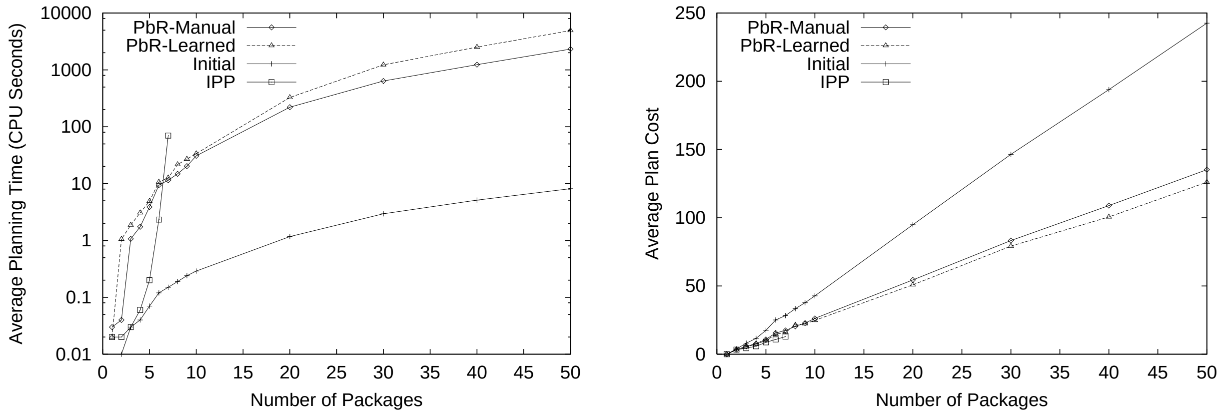
```
(define-rule :name logs-1
  :if (:operators
        ((?n1 (drive-truck ?t ?l1 ?l2 ?c))
         (?n2 (drive-truck ?t ?l2 ?l1 ?c))))
  :replace (:operators ((?n1 ?n2)))
  :with NIL)

(define-rule :name logs-2
  :if (:operators
        ((?n1 (drive-truck ?t ?l1 ?l2 ?c))))
  :replace (:operators ((?n2)))
  :with (:operators
          ((?n2 (drive-truck ?t ?l3 ?l2 ?c)))))

(define-rule :name logs-3
  :if (:operators
        ((?n1 (drive-truck ?t ?l1 ?l2 ?c))
         (?n2 (drive-truck ?t ?l2 ?l3 ?c)))
       :links ((?n1 (at ?t ?l2) ?n2)))
  :replace (:operators ((?n1 ?n2)))
  :with (:operators
          ((?n3 (drive-truck ?t ?l1 ?l3 ?c)))))
```

Figure 29: Learned Rewriting Rules: Logistics

IPP could only solve problems up to 7 packages (it also solved 10 out of 20 for 8 packages, and 1 out of 20 for 9 packages, but these are not shown in the figure). Figure 30(a) shows the average planning time. Figure 30(b) shows the average cost for the 50 packages range. The results are similar to the previous experiments. Initial is efficient but highly suboptimal. PbR is able to considerably improve the cost of this plans and approach the optimal. Most interestingly, the learned rules in this domain achieve better quality plans than the manual ones. The reason is the more general nature of learned `logs-1` and `logs-3` rules compared to the manual `loop` and `triangle` rules.

6 Related Work

PbR is designed to find a balance among the requirements of planning efficiency, high quality plans, flexibility, and extensibility. A great amount of work on AI Planning has focused on improving its average-case efficiency given that the general cases are computationally hard (Bylander, 1994; Erol et al., 1995). Often, this is achieved by incorporating domain knowledge either manually-specified by experts (e.g., Bacchus & Kabanza, 2000) or automatically learned search control (e.g., Minton, 1988; Etzioni, 1993; Gerevini & Schubert, 1998; Fox & Long, 1998; Rintanen, 2000). Although all these approaches do improve the efficiency of planning, they do not specifically address plan quality, or else they consider only very simple cost metrics (such as the number of steps). Some systems learn search control that addresses both planning efficiency and plan quality (Estlin & Mooney, 1997; Borrajo & Veloso, 1997; Pérez, 1996). However, from the reported experimental results, PbR appears to be more



(a) Average Planning Time (b) Average Plan Cost

Figure 30: Performance with Learned Rules: Logistics

scalable. Moreover, PbR provides an anytime algorithm while other approaches must run to completion.

Local search has a long tradition in combinatorial optimization (Aarts & Lenstra, 1997; Papadimitriou & Steiglitz, 1982). Local improvement ideas have found application in many domains. Some of the general work most relevant to PbR is on constraint satisfaction (e.g., the min-conflicts heuristic: Minton, 1992), satisfiability testing (e.g., GSAT: Selman, Levesque, & Mitchell, 1992), and scheduling (Zweben, Daun, & Deale, 1994). Our work is inspired by these approaches but there are several differences. First, PbR operates on complex graph structures (partial-order plans) as opposed to variable assignments. Second, our repairs are declaratively specified and may be changed for each problem domain, as opposed to their fixed, generic repair strategies. Third, PbR accepts arbitrary measures of quality, not just constraint violations as in min-conflicts, or number of unsatisfied clauses as GSAT. Finally, PbR searches the space of valid solution plans, as opposed to the space of variable assignments which may be internally inconsistent.

PbR builds on ideas from graph rewriting (Schürr, 1997). The plan-rewriting rules in PbR are an extension of traditional graph rewriting rules. By taking advantage of the semantics of planning, PbR introduces partially-specified plan-rewriting rules, where the rules do not need to specify the completely detailed embedding of the consequent as in pure graph rewriting. Nevertheless, there are several techniques that can transfer from graph rewriting into Planning by Rewriting, particularly for fully-specified rules. Dorr (1995) defines an abstract machine for graph isomorphism and studies a set of conditions under which traditional graph rewriting can be performed efficiently. Perhaps a similar abstract machine for plan rewriting can be defined. The idea of rule programs also appears in this field and has been implemented in the PROGRES system (Schürr, 1997).

The work most closely related to our plan-rewriting algorithm is plan merging (Foulser, Li, & Yang, 1992). Foulser et al. provide a formal analysis and algorithms for exploiting positive interactions within a plan or across a set of plans. However, their work only considers the case in which a set of operators can be replaced by *one* operator that provides the same effects to the rest of the plan and consumes the same or fewer preconditions. Their focus is on optimal and approximate algorithms for this type of operator merging. Plan rewriting in PbR can be seen as a generalization of operator merging where a subplan can replace another subplan. A difference is that PbR is not concerned with finding the optimal merge (rewritten plan) in a single pass of an optimization algorithm as their approach does. In PbR we are interested in generating possible plan rewritings during each rewriting phase, not the optimal one. The optimization occurs as the (local) search progresses.

Case-based planning (e.g., Kambhampati, 1992; Veloso, 1994; Nebel & Koehler, 1995; Hanks & Weld, 1995) solves a problem by modifying a previous solution. There are two phases in case-based planning. The first one identifies a plan from the library that is most similar to the current problem. In the second phase this previous plan is adapted to solve the new problem. PbR modifies a solution to the current problem, so there is no need for a retrieval phase nor the associated similarity metrics. Plan rewriting in PbR can be seen as a type of adaptation from a solution to a problem to an alternate solution for the same problem. That is, a plan rewriting rule in PbR identifies a pair of subplans (the replaced and replacement subplans) that may be interchangeable.

Plan rewriting has been applied to several real world domains. Autominder (Pollack, Brown, Colbry, McCarthy, Orosz, Peintner, Ramakrishnan, & Tsamardinos, 2003) is a comprehensive system to assist the elderly with declining cognitive functions that is embodied in the nursing robot Pearl (Pineau, Montemerlo, Pollack, Roy, & Thrun, 2003). The personalized cognitive orthotic (PCO) (McCarthy & Pollack, 2002) of Autominder uses plan rewriting techniques to create reminder plans for elderly patients and to update these plans in response to environment changes. A PbR-based query planner for data integration is described in (Ambite & Knoblock, 2000).

Our approach to learning plan-rewriting rules is closely related to learning search control. In a sense, our plan rewriting rules can be seen as "a posteriori" search control. Instead of trying to find search control that would steer the planner during generation towards the optimal plan and away from fruitless search, our approach is to generate fast a suboptimal initial plan, and then optimize it, after the fact, by means of the rewriting rules.

Our rule generalization algorithm has some elements from Explanation-Based Learning (EBL) (Minton, 1988; Kambhampati, Katukam, & Qu, 1996; Estlin & Mooney, 1996), but it compares two complete plans, with the aid of the operator specification, as opposed to problem-solving traces. Similarly to EBL search control rules, our learned plan rewriting rules also suffer from the utility problem (Minton, 1988).

Search control can also be learned by analyzing the operator specification without using any examples (Etzioni, 1993). Similar methods could be applied to PbR. For example, we could systematically generate rewriting rules that replace a set of operators by another set that achieves similar effects. Then, test the rules empirically and select those of highest utility. Upal (2001) presents techniques to learn rewriting rules by static domain analysis and by analysis of problem solving traces.

Upal & Elio (2000) compare the performance of search control rules versus plan rewriting rules (both learned from problem solving traces). In their experiments, search control rules are more effective than rewriting rules. However, it is unclear whether this is due to their specific rule learning algorithm or to some intrinsic limitation of plan rewriting, since they do not report the number or types of the learned rewriting rules, nor evaluate their utility (cf.

(Minton, 1988)). Thus, the relative merit of learning rewriting rules versus search control remains an open problem.

In scheduling, several learning techniques have been successfully applied to obtain search control for iterative repair. Zweben et al. (Zweben, Davis, Daun, Drascher, Deale, & Eskey, 1992) used an extension of EBL to learn the utility of the repairs, selecting when to apply a more-informed versus less-informed repair. Zhang and Dietterich (1995) used a reinforcement learning approach to select repair strategies for the same problem. Both systems learn how to select the repairs to improve the efficiency of search, but they do not learn the repairs themselves as in our work.

7 Discussion and Future Work

In this chapter, we have presented Planning by Rewriting, a paradigm for efficient high-quality planning. PbR adapts graph rewriting and local search techniques to the semantics of domain-independent partial-order planning. The basic idea of PbR consists in transforming an easy-to-generate, but possibly suboptimal, initial plan into a high-quality plan by applying declarative plan-rewriting rules in an iterative repair style.

There are several important advantages to the PbR planning approach. First, PbR is a *declarative domain-independent* framework, which brings the benefits of reusability and extensibility. Second, it addresses *sophisticated plan quality* measures, while most work in domain-independent planning has not addressed quality or does it in very simple ways. Third, PbR is *scalable* because it uses efficient local search methods. In fact, PbR provides domain-independent framework for local search. Finally, PbR is an *anytime* planning algorithm that allows balancing planning effort and plan quality in order to maximize the utility of the planning process.

An open area of research is to relax our framework to accept incomplete plans during the rewriting process. This expands the search space considerably and some of the benefits of PbR, such as its anytime property, are lost. But for some domains the shortest path of rewritings from the initial plan to the optimal may pass through incomplete or inconsistent plans. This idea could be embodied as a planning style that combines the characteristics of generative planning and Planning by Rewriting. This is reminiscent of the plan critics approach (Sacerdoti, 1975; Sussman, 1975). The resulting plan-rewriting rules can be seen as declarative specifications for plan critics. The plan refinements of both partial order planning (Kambhampati et al., 1995) and Hierarchical Task Network Planning (Erol, Nau, & Hendler, 1994) can be easily specified as plan-rewriting rules.

Planning by Rewriting is also well suited to mixed-initiative planning. In mixed-initiative planning, the user and the planner interact in defining the plan. For example, the user can specify which are the available or preferred actions at the moment, change the quality criteria of interest, etc. In fact, some domains can only be approached through mixed-initiative planning. For example, when the quality metric is very expensive to evaluate, such as in geometric analysis in manufacturing, the user must guide the planner towards good quality plans in a way that a small number of plans are generated and evaluated. Another example is when the plan quality metric is multi-objective or changes over time. Several characteristics of PbR support mixed-initiative planning. First, because PbR offers complete plans, the user can easily understand the plan and perform complex quality assessment. Second, the rewriting rule language is a convenient mechanism by which the user can propose modifications to the plans. Third, by selecting which rules to apply or their order of application the user can guide the planner.

We plan to develop a system that can automatically learn the optimal planner configuration for a given domain and problem distribution in a manner analogous to Minton's Multi-TAC system (Minton, 1996). Such system would perform a search in the configuration space of the PbR planner proposing different initial plan generators, candidate sets of rewriting rules, and search methods. By testing each proposed configuration against a training set of simple problems, the system would hill-climb in the configuration space in order to achieve the most useful combination of rewriting rules and search strategy.

The PbR framework achieves a balance between domain knowledge, expressed as plan-rewriting rules, and general local-search techniques that have proved useful in many hard combinatorial problems. We expect that these ideas will push the frontier of solvable problems for many practical domains in which high quality plans and anytime behavior are needed.

Acknowledgments

This chapter is based on (Ambite & Knoblock, 2001; Ambite et al., 2000; Ambite, 1998).

The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-1-0504, in part by the National Science Foundation under grant numbers IRI-9313993, IRI-9610014 in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract numbers F30602-94-C-0210, F30602-97-2-0352, F30602-97-2-0238, F30602-98-2-0109, in part by the United States Air Force under contract number F49620-98-1-0046, in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152, and in part by a Fulbright/Ministerio of Educación y Ciencia of Spain scholarship.

The U.S.Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

Aarts, E., & Lenstra, J. K. (1997). *Local Search in Combinatorial Optimization.* John Wiley and Sons, Chichester, England.

Ambite, J. L. (1998). *Planning by Rewriting.* Ph.D. thesis, University of Southern California.

Ambite, J. L., & Knoblock, C. A. (2000). Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence*, *118*(1-2), 115–161.

Ambite, J. L., & Knoblock, C. A. (2001). Planning by Rewriting. *Journal of Artificial Intelligence Research*, *15*, 207–261.

Ambite, J. L., Knoblock, C. A., & Minton, S. (2000). Learning plan rewriting rules. In *Proceedings of the Fifth International Conference on Artificial Intelligence on Planning and Scheduling* Breckenridge, CO.

Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That.* Cambridge University Press.

Bacchus, F., & Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, *116*(1–2), 123–191.

Bäckström, C. (1994). Executing parallel plans faster by adding actions. In Cohn, A. G. (Ed.), *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pp. 615–619 Amsterdam, Netherlands. John Wiley and Sons.

Blum, A. L., & Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*(1–2), 281–300.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1-2), 5–33.

Borrajo, D., & Veloso, M. (1997). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review*, *11*, 371–405.

Bylander, T. (1994). The computation complexity of propositional strips. *Artificial Intelligence*, *69*(1-2), 165–204.

Dean, T., & Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 49–54 Saint Paul, MN.

Dorr, H. (1995). *Efficient graph rewriting and its implementation*, Vol. 922 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA.

Erol, K., Nau, D., & Hendler, J. (1994). UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pp. 249–254 Chicago, IL.

Erol, K., Nau, D., & Subrahmanian, V. S. (1995). Decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, *76*(1-2), 75–88.

Estlin, T. A., & Mooney, R. J. (1996). Multi-strategy learning of search control for partial-order planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 843–848 Portland, OR.

Estlin, T. A., & Mooney, R. J. (1997). Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 1227–1233 Nagoya, Japan.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, *62*(2), 255–302.

Etzioni, O., & Weld, D. S. (1994). A softbot-based interface to the Internet. *Communications of the ACM*, *37*(7).

Foulser, D. E., Li, M., & Yang, Q. (1992). Theory and algorithms for plan merging. *Artificial Intelligence*, *57*(2–3), 143–182.

Fox, M., & Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificicial Intelligence Research*, *9*, 367–421.

Gerevini, A., & Schubert, L. (1998). Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pp. 905–912 Madison, WI.

Glover, F. (1989). Tabu search–Part I. *ORSA Journal on Computing*, *1*(3), 190–206.

Gupta, N., & Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, *56*(2–3), 223–254.

Hanks, S., & Weld, D. S. (1995). A domain-independent algorithm for plan adaptation. *Journal of Artificicial Intelligence Research*, *2*, 319–360.

Kambhampati, S. (1992). A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, *55*(2-3), 193–258.

Kambhampati, S., Katukam, S., & Qu, Y. (1996). Failure driven dynamic search control for partial order planners: an explanation based approach. *Artificial Intelligence*, *88*(1-2), 253–315.

Kambhampati, S., Knoblock, C. A., & Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating the design tradeoffs in partial order planning. *Artificial Intelligence*, *76*(1-2), 167–238.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, *68*(2), 243–302.

Knoblock, C. A. (1996). Building a planner for information gathering: A report from the trenches. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems* Edinburgh, Scotland.

Koehler, J., & Hoffmann, J. (2000). On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, *12*, 338–386.

Koehler, J., Nebel, B., Hoffman, J., & Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proceedings of the Fourth European Conference on Planning (ECP-97)*, pp. 273–285.

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, *21*, 498–516.

McCarthy, C. E., & Pollack, M. E. (2002). Plan-based personalized cognitive orthotic. In *Proceedings of the Sixth International Conference on Artificial Intelligence on Planning and Scheduling* Toulouse, France.

Minton, S. (1988). *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Ph.D. thesis, Computer Science Department, Carnegie Mellon University.

Minton, S. (1992). Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, *58*(1-3), 161–205.

Minton, S. (1996). Automatically configuring constraint satisfaction programs: A case study. *Constraints*, *1*(1), 7–43.

Minton, S., & Underwood, I. (1994). Small is beautiful: A brute-force approach to learning first-order formulas. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 168–174 Seattle, WA.

Nau, D. S., Gupta, S. K., & Regli, W. C. (1995). AI planning versus manufacturing-operation planning: A case study. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* Montreal, Canada.

Nebel, B., & Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, *76*(1-2), 427–454.

Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ.

Penberthy, J. S., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Third International Conference on Principles of Knowledge Representation and Reasoning*, pp. 189–197 Cambridge, MA.

Pérez, M. A. (1996). Representing and learning quality-improving search control knowledge. In *Thirteenth International Conference on Machine Learning* Bari, Italy.

Pineau, J., Montemerlo, M., Pollack, M., Roy, N., & Thrun, S. (2003). Towards robotic assistants in nursing homes: Challenges and results. *Robotics and Autonomous Systems*, *42*(3-4), 271–281.

Pollack, M. E., Brown, L., Colbry, D., McCarthy, C. E., Orosz, C., Peintner, B., Ramakrishnan, S., & Tsamardinos, I. (2003). Autominder: An intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems*, *44*(3-4), 273–282.

Rintanen, J. (2000). An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence* Austin, TX.

Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 206–214 Tbilisi, USSR.

Schürr, A. (1997). Programmed graph replacement systems. In Rozenberg, G. (Ed.), *Handbook on Graph Grammars: Foundations*, Vol. 1, pp. 479–546. World Scientific, Singapore.

Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 440–446 San Jose, California. AAAI Press.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (1997). *Database System Concepts* (Third edition). McGraw-Hill.

Slaney, J., & Thiébaux, S. (1996). Linear time near-optimal planning in the blocks world. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pp. 1208–1214 Menlo Park. AAAI Press / MIT Press.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. American Elsevier, New York.

Upal, M. A. (2001). Learning plan rewrite rules using static domain analysis. In *Proceedings of the Fourteenth International FLAIRS Conference*, pp. 412–417 Key West, Florida.

Upal, M. A., & Elio, R. (2000). Learning search control rules versus rewrite rules to improve plan quality. In *Proceedings of the Thirteenth Canadian Conference on Artificial Intelligence*, pp. 240–253 Montréal, Québec, Canada.

Veloso, M. (1994). *Planning and Learning by Analogical Reasoning*. Springer Verlag.

Yu, C., & Chang, C. (1984). Distributed query processing. *ACM Computing Surveys*, *16*(4), 399–433.

Zhang, W., & Dietterich, T. C. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1114–1120 Montreal, Canada.

Zweben, M., Daun, B., & Deale, M. (1994). Scheduling and rescheduling with iterative repair. In *Intelligent Scheduling*, pp. 241–255. Morgan Kaufman, San Mateo, CA.

Zweben, M., Davis, E., Daun, B., Drascher, E., Deale, M., & Eskey, M. (1992). Learning to improve constraint-based scheduling. *Artificial Intelligence*, *58*(1–3), 271–296.