In *Proceedings of the Fourteenth National Conference on Artificial Intelligence, Providence, Rhode Island, 1997.*

Planning by Rewriting: Efficiently Generating High-Quality Plans*

José Luis Ambite and **Craig A. Knoblock**
Information Sciences Institute and Department of Computer Science
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292
{ambite, knoblock}@isi.edu

Abstract

Domain-independent planning is a hard combinatorial problem. Taking into account plan quality makes the task even more difficult. We introduce a new paradigm for efficient high-quality planning that exploits plan rewriting rules and efficient local search techniques to transform an easy-to-generate, but possibly suboptimal, initial plan into a low-cost plan. In addition to addressing the issues of efficiency and quality, this framework yields a new anytime planning algorithm. We have implemented this planner and applied it to several existing domains. The results show that this approach provides significant savings in planning effort while generating high-quality plans.

Introduction

Planning is the process of generating a network of actions that achieves a desired goal from an initial state of the world. Domain independent planning accepts as input, not only the initial state and the goal, but also the domain specification (i.e., the operators). This is a problem of considerable practical significance, but domain-independent planning is computationally hard except for its simplest formulations (Erol, Nau, & Subrahmanian 1995). Moreover, in many circumstances it is not enough to find any solution plan since the quality of the solution is important. This paper presents a new paradigm for efficiently generating high-quality plans.

Two observations guided the present work. First, there are two sources of complexity in planning:

- Satisfiability: the difficulty of finding any solution to a planning problem.
- Optimization: the difficulty of finding the optimal solution according to 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 easy and their complexity is dominated by the optimization problem. For example, there may be many plans that would solve the problem, so finding one is simple (that is, in polynomial time), but the cost of each solution varies greatly so that finding the optimal one may be difficult. We shall refer to these domains as optimization domains. Some optimization domains of great practical interest are query access planning and process planning.[1]

Second, planning problems have a great deal of structure. Plans are a type of graphs with strong semantics, determined both by 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 easy. 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 plan rewriting rules until either an acceptable solution is found or a resource limit is reached.

There are several important points to note in this basic framework. First, the rewritten plans are always solutions to the given planning problem. Thus, the search occurs in the space of solution plans, which is in many cases much smaller than the space of partial plans that other planning systems usually explore. Second, efficient search of the space of rewritings is critical to the success of the method. Thus, we adapt techniques from local search to help in this process. Finally, our framework yields an anytime algorithm (Dean & Boddy 1988). The planner always has a solution to offer at any point in its computation (modulo the initial plan generation, which should 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

*Copyright ©1997, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

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

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.

As motivation, consider two domains: query processing in a distributed, heterogeneous environment and manufacturing process planning. Distributed query processing (Yu & Chang 1984) involves generating a plan that efficiently computes a user query. This plan is composed of data retrieval actions at diverse information sources and operations on this data (such as join, selection, etc). Some systems use a general-purpose planner to solve this problem (Knoblock 1996). In this domain it is relatively easy to construct an initial plan 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) 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.

`join-swap`
$get(q1, db1) \bowtie (get(q2, db2) \bowtie get(q3, db3)) \Leftrightarrow$
$get(q2, db2) \bowtie (get(q1, db1) \bowtie get(q3, db3))$
`remote-join-eval`
$(get(R, db) \bowtie get(S, db)) \wedge capability(db, join) \Rightarrow$
$get(R \bowtie S, db)$

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 possible 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 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. Such grouping of machining operations on a setup is analogous to evaluating a subquery at a remote information source.

In summary, this paper develops a new planning paradigm yielding several contributions. First, by using local search techniques, high-quality plans can be efficiently generated. Second, the rewriting rules provide a natural and convenient mechanism to specify complex plan transformations. Third, it offers a new anytime planning algorithm.

Planning by Rewriting

We will describe the main issues in Planning by Rewriting as an instantiation of the local search idea (Papadimitriou & Steiglitz 1982):

- *Selection of an initial feasible point*: How to efficiently generate an initial solution plan.
- *Generation of a local neighborhood*: The neighborhood is the set of plans obtained from the application of the plan rewriting rules.
- *Cost function to minimize*: The given plan evaluation function could range from a simple domain independent cost metric, such as the number of steps, to more complex domain specific ones, such as query evaluation cost or manufacturing time for a set of parts.
- *Selection of the next point*: What is the next plan to consider. This choice determines how the global space will be explored and has a significant impact on the efficiency of planning. For example, steepest descent, first improvement, random walk, etc.

In the following subsections we expand these topics. First, we introduce some background on planning and rewriting. Second, we discuss the initial plan generation. Third, we show how the local neighborhood is generated by the rewriting rules and present their syntax, their semantics, and a rule taxonomy. Finally, we address the selection of the next plan.

Planning and Rewriting Concepts

A plan is represented by a graph notation, in the spirit of partial-order causal-link (POCL) planners, such as UCPOP (Penberthy & Weld 1992). The nodes are plan steps, that is, domain actions. The edges specify a temporal ordering relation among steps, imposed by causal links and ordering constraints. A causal link is a record of how a condition is used in a plan. This record contains the condition, a step that produces (establishes) it, and a step that consumes it (that is, a step which needs it as a precondition). By causality, the producer must precede the consumer. The ordering constraints arise from solving operator threats and resource conflicts. An operator threat occurs when a step has an effect that negates the condition of a causal link and can possibly be ordered between its producer and its consumer. To prevent this situation, which possibly makes the plan inconsistent, POCL planners order the threatening step either before the producer (promotion) or after the consumer (demotion).

Operators may need to use certain resources to perform their actions. In this paper, we consider unit non-consumable resources, that is, those that are fully acquired by an operator until the completion of its action, and then released to be reused (Knoblock 1994b). For this type of resource, steps requiring the same resource have to be sequentially ordered. Finally, note

that all conditions in the plan are fully ground because we start with a complete initial plan.

A plan rewriting rule, akin to term and graph rewriting rules, specifies the replacement under certain conditions of a partial plan by another partial plan. Our system ensures that the rewritten plan remains complete and consistent. These rules are intended to improve the quality of the plans.

Generation of an Initial Plan

Fast initial plan generation is domain-specific in nature. It requires the user to specify an efficient mechanism to compute the initial solution plan. By the definition of optimization domains this should not be hard. We have experimented with two approaches to construct feasible initial plans: using a planner with search control rules and exploiting simple domain-specific approximation algorithms.

A very general way of efficiently constructing plans is to use a domain-independent generative planner that accepts search control rules. 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 we used depth-first search and a goal selection heuristic based on abstraction hierarchies (Knoblock 1994a). This combination quickly generates a feasible plan, but often the time required to manufacture all objects is suboptimal.

For many domains, we expect that simple domain-dependent greedy algorithms will provide good initial plans. For example, in the query planning domain, the system can easily generate initial query evaluation plans by parsing the given query. In the blocksworld it is also straightforward to generate a solution in linear time using the naive algorithm: put all blocks on the table and build the desired towers from the bottom up.

Generation of a Local Neighborhood

The plan rewriting rules determine the neighborhood of the current plan to be explored. They embody the domain-specific knowledge about what transformations of a solution plan are likely to result in higher-quality solutions. In this section we describe the syntax and the semantics of the rules, as well a taxonomy of plan rewriting rules.

Rule Syntax and Semantics First, we introduce the rule syntax and semantics through some examples. Then, we provide a formal description. A sample rule in the blocks world domain is given in Figure 2. Intuitively, it says that, whenever possible, it is better to stack a block on top of another directly, rather than first moving it to the table.

```
(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))))
```

Figure 2: Blocks World Rewriting Rule[2]

A rule for a manufacturing domain (Minton 1988) is shown in Figure 3. It states that if a plan includes two consecutive punching operations to make holes in two different objects, but another machine, a drill-press, is also available, the plan can be parallelized by replacing one of the punch operations by using the drill-press.

```
(define-rule :name punch-by-drill-press
  :if (:operators ((?n1 (punch ?o1 ?width1 ?orn1))
                   (?n2 (punch ?o2 ?width2 ?orn2)))
       :links (?n1 ?n2)
       :constraints ((:neq ?o1 ?o2)
                     (possibly-adjacent ?n1 ?n2)))
  :replace (:operators (?n1))
  :with (:operators
          (?n3 (drill-press ?o1 ?width1 ?orn1))))
```

Figure 3: Process Planning Rewriting Rule

In general, the rule syntax follows the template in Figure 4. The rewriting algorithm is outlined in Figure 5. The semantics of the rules is as follows. The antecedent, the `:if` field, describes a graph specification (operators, links, and constraints) that is matched against the plan. The `:operators` field consists of a list of step number and step predicate pairs. Each step predicate is interpreted as an step action (or as one of the resources used by the step, if the keyword `:resource` is present, e.g. Figure 7). The `:links` field consists of a list of link specifications. A link specification can match either any ordering link in the plan, a causal link if a predicate is given, or an ordering link introduced in the resolution of threats (if the keyword `:threat` is present). Finally, built-in and user-defined predicates can be specified in the `:constraints` field. The built-in predicates include inequalities (`:neq`), comparison, and arithmetic predicates. The user-defined predicates may act as filters on the previous variables or introduce new variables (and compute new values for them). Formally, the language of the antecedent forms a conjunctive query with interpreted predicates against the plan graph. The rule matches can be computed either all at the same time, as in bottom-up evaluation of logic databases, or one-

[2] Note that the link expression in the antecedent is actually redundant. Unstack puts the block `?b1` on the table from where it is picked up by the stack operator, so that the causal link is already implied by the `:operators` and `:constraints` specification. The interpreted predicate `possibly-adjacent` ensures that the operators are consecutive in some linearization of the plan.

at-a-time as in Prolog. Which option is preferable depends on the search strategy.

```
(define-rule :name <rule-name>
 :if (:operators ((<nv> <np> {:resource}) ...)
      :links ((<nv> {<lp>|:threat} <nv>) ...)
      :constraints (<ip> ...))
 :replace (:operators (<nv> ...)
           :links ((<nv> {<lp>|:threat} <nv>) ...))
 :with (:operators ((<nv> <np> {:resource}) ...)
        :links ((<nv> {<lp>} <nv>) ...)))

 <nv> = node variable, <np> = node predicate,
 <lp> = causal link predicate, {} = optional
 <ip> = interpreted predicate,  | = alternative
```

Figure 4: Rewriting Rule Template

1. Match rule antecedent, `:if` field, against the plan, returning a set of candidate rule instantiations.
2. For each antecedent instantiation:
   (a) Remove from the plan the subgraph specified in the `:replace` field.
   (b) Generate all *consistent* embeddings of the subgraph specified in the `:with` field.

Figure 5: Outline of Plan Rewriting Algorithm

Rules must be safe, that is, all the variables appearing in the consequent of the rules, `:replace` and `:with` fields, have to appear in the antecedent. The `:replace` field identifies the subgraph that is going to be removed from the plan (a subset of steps and links of the antecedent). The `:with` field specifies the replacement subgraph. The system generates all valid embeddings of the replacement subplan into the original plan (once the subplan in the `:replace` field has been removed). Thus, a single rule instantiation may produce several rewritten plans. The formal conditions for valid rewriting, a generalization from plan merging in (Foulser, Li, & Yang 1992), are shown in Figure 6. It is possible to define rules whose application provably yields a correct plan. However, this *eager* approach would require the generation of many rules with very long and specific antecedents, which are possibly expensive to match. An alternative is a *lazy* approach in which the rule antecedents only include a subset of the conditions necessary for a valid rewriting. In this case, when the rules are applied, the rewritten plans are checked for correctness. The "lazy" approach allows the specification of more natural rules that express the main idea of the transformation instead of focusing on technicalities or rare cases. We used the latter for our experiments.

A subplan $S1$, embedded in a plan P, can be replaced by a subplan $S2$, resulting in plan P', iff there exists an ordering O, such that $P' = (P - S1) \cup S2 \cup O$ is a consistent plan, and $\text{NetPreconditions}(S2, P') \subseteq \text{NetPreconditions}(S1, P)$, and $\text{UsefulEffects}(S1, P) \subseteq \text{UsefulEffects}(S2, P')$.

Useful Effects of a subplan S, embedded in a plan P, are those conditions present in causal links whose producer is in S and whose consumer is in $P - S$.

Net Preconditions of a subplan S, embedded in a plan P, are those conditions in causal links whose consumer is in S and whose producer is in $P - S$.

Figure 6: Conditions for Valid Rewriting

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 that may automatically deduce the rules from the domain specification (see Future Work), it is helpful to know what kinds of rules are useful. So far 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, and the `join-swap` rule in Figure 1 that combines the commutative and associative properties of the relational algebra.

```
(define-rule :name resource-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: Reorder Rewriting Rule

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, which prefers to evaluate a join between two tables that come from the same source at the remote source rather than locally (if the source has join processing capabilities). Another example is the blocksworld rule in Figure 2.

Expand: These are rules that do the inverse of collapse. Although we did not find this rule type in the domains analyzed so far, 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. For example, when some of these operators are already present in the plan and can be synergistically reused.

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 3.

Selection of Next Plan

The strategy to select the next plan to consider determines the way the solution space is searched. The rules generate the "natural perturbations" of a plan, but which rewriting, if any, will lead towards the global optimum cannot be predicted in general. We have explored gradient descent techniques, such as first improvement and steepest descent. In *first improvement*, the next plan to consider is the first rewriting that improves the cost. This has the advantage that the neighborhood is generated only up to the point such a plan is found, but the improvement may not be the best that could be achieved in that neighborhood. In *steepest descent*, the minimum cost plan within the neighborhood is chosen. This guarantees the biggest improvement in cost in each iteration, but it requires the whole neighborhood to be searched.

In general, the space of rewritings and the cost functions are not convex, thus our gradient descent techniques can get caught in local minima. To move towards the optimum escaping low-quality local minima, we used two techniques: restart and random walk. In the first one, the system restarts the rewriting process a fixed number of times from a different initial plan. This technique requires an initial plan generator that is able to provide several different/random initial plans. The second technique is applied when the local minima are not strict and consists of a random walk of a fixed length along the plateau.

Initial Results

We have implemented the planner described in this paper and applied it in several different application domains. In this section we report on our initial results in the domains of manufacturing process planning and distributed query planning.

Process planning

The task in the manufacturing process planning domain is to find a plan to manufacture a set of parts. We implemented the domain specification in (Minton 1988). This domain contains a variety of machines, such as a lathe, punch, spray painter, welder, etc, which are used to perform various operations to produce a set of parts. In this domain all of the machining operations are assumed to take unit time and the optimal plan is the one that requires the minimal length schedule. There are ten possible machining operations for making a part. Sample rewriting rules for this domain appear in Figures 3 and 7.

To evaluate the performance of Planning by Rewriting (PBR), we compared it to a planner called Sage (Knoblock 1995), which is an extension of UCPOP that supports resources, execution and replanning. For PBR, we defined ten plan rewriting rules for this domain and used a steepest descent search. We ran Sage with a best-first search over the length of the schedule (Sage-BFS) in order to find the optimal plan. We used Sage with depth-first search and a goal selection heuristic based on abstraction hierarchies (Sage-DFS) to generate plans as fast as possible. Sage-DFS is also used for the initial plan generator for PBR. We tested each of the three systems on 300 problems that ranged from 1 to 12 goals (25 in each set). There were 80 provably unsolvable problems. Sage-DFS was able to solve 22 more problems than Sage-BFS in the given search limit of 50,000 nodes (Sage-DFS proved 16 solvable and 6 unsolvable). The plan size (number of steps) grows linearly with the number of goals, from 3 steps for the one goal problems to 14 steps for the 12 goal problems.

The results are shown in Figures 8 and 9. Figure 8 shows the average time on the solvable problems for each problem set in the three configurations. Figure 9 shows the average schedule length for the problems solved by all planners. As shown in the graphs, PBR takes slightly longer than Sage-DFS, as expected since Sage-DFS is also used to generate the initial plans for PBR, but is able to improve their quality significantly. PBR performs about the same as Sage-BFS on the easy problems, both in time and in quality. For harder problems, PBR is much more efficient than Sage-BFS and it produces plans whose quality is close to the optimal. These results show the benefits of finding an suboptimal initial plan quickly and then efficiently transforming it to improve its quality as proposed in PBR.

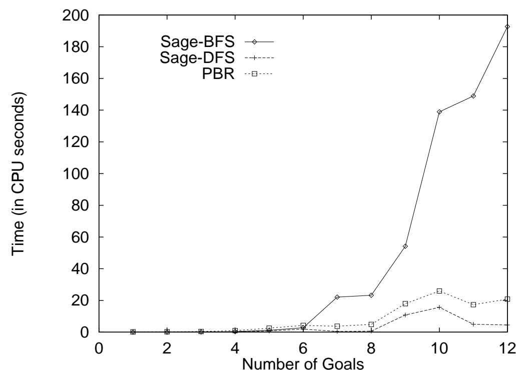

Figure 8: Manufacturing Time Comparison

Query planning

Distributed query processing involves generating a plan that efficiently computes a user query. This plan is composed of data retrieval actions at diverse information sources and operations on this data. We used a simplified domain for the query planner (Sage) of the SIMS mediator (Arens, Knoblock, & Shen 1996). We compare the performance and quality of the Sage planner and PBR for this domain, where the query plans are trees of join operations.

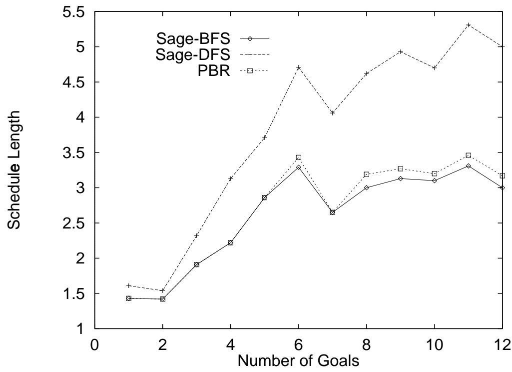

Figure 9: Schedule Length Comparison

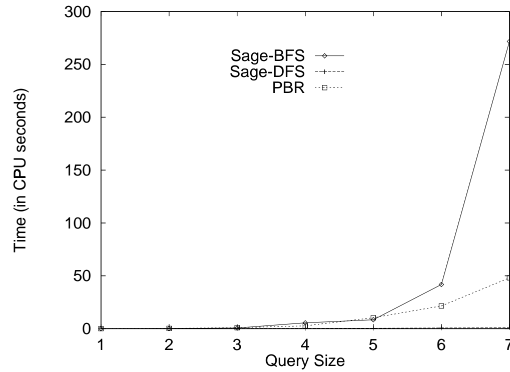

Figure 10: Query Planning Time Comparison

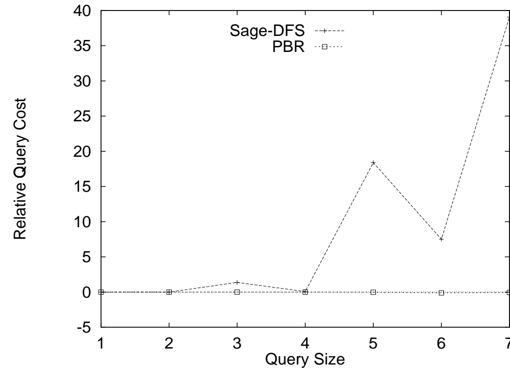

Figure 11: Query Plan Quality Comparison

In the query planning domain, Sage performs a best-first search with a heuristic commonly used in query optimization that explores only the space of left join trees (Sage-BFS). For PBR, we defined the `join-swap` rule of Figure 1. The initial plans were random depth-first search parses of the query (Sage-DFS). To escape local minima, PBR generates and rewrites three random initial plans and picks the best rewriting. The cost metric for all planners is based on an estimation of the cost of the join operations and the size of the intermediate results transmitted between the sources and the mediator. In this experiment, we used a set of 43 conjunctive queries previously defined for a logistics planning application involving from one to seven relations. All queries could be solved by all planners. A set of eight relation joins could not be solved by Sage within 50,000 nodes, while PBR could easily solve them with low cost plans.

The results are shown in Figures 10, 11, and 12. Figure 10 shows the average time for each query set for query sizes from one to seven. The times for PBR includes both the generation of the three random initial plans and their rewriting. Figure 11 shows the average quality of Sage-DFS and PBR normalized with respect to Sage-BFS. The normalization is done for clarity because the values for Sage-BFS query cost vary considerably and we want to show how PBR and Sage-DFS perform relative to Sage-BFS as the problem size increases. The graph shows that Sage-DFS produces very poor quality plans on the large problems. Figure 12 shows in more detail the average quality of PBR normalized with respect to Sage-BFS. This graph shows that PBR performs as well as Sage-BFS for the smaller queries and even finds better solutions for the larger ones (up to 12% better). This is not surprising because of the restricted space of left trees that Sage-BFS is searching. As in the manufacturing domain, PBR shows better scaling properties than the corresponding systematic algorithms.

Related work

Some of the most closely related work is on plan merging (Foulser, Li, & Yang 1992). Their system solves a complex goal by dividing it into subgoals, solving the subproblems, and combining the partial solutions exploiting synergies. They improve the quality of a plan by replacing a set of operators by *one* operator that can do the same job. Planning by Rewriting differs in that it starts with a complete solution plan for the original goal, and it generalizes plan merging by allowing the replacement of a subplan by another subplan, thus expanding the types of plan transformations and the opportunities for cost reduction.

Case-based planning also attempts to solve a problem by modifying a previous solution (Veloso 1994; Paulokat & Wess 1994). Systematic algorithms, such as (Hanks & Weld 1995), invert the decisions done in refinement planning to find a path between the solution to a similar old problem and the new problem. Our work modifies a solution to the current problem, so there is no need for similarity metrics, nor retrieval process. Moreover, our rewriting rules indicate how to transform a solution into another solution plan, rather than searching blindly up and down the space of partial plans. However, the rules in PBR may search the space of rewritings non-systematically. Such an effect

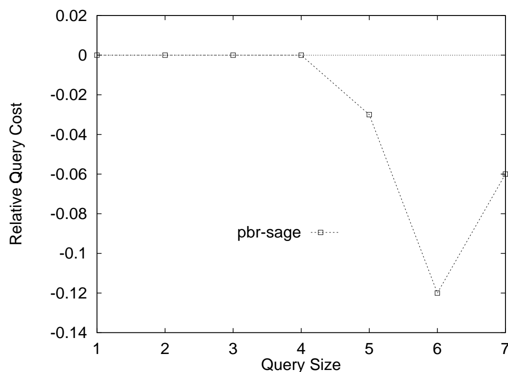

Figure 12: Query Plan Quality (PBR only)

is ameliorated by the gradient-descent search strategy.

Local search has a long tradition in combinatorial optimization (Papadimitriou & Steiglitz 1982). Local improvement ideas have found application in constraint satisfaction, scheduling, and heuristic search. In constraint satisfaction, (Minton 1992) start with a complete, but inconsistent, variable assignment and efficiently search the space of repairs using a simple heuristic, min-conflicts. In our work we focus on a STRIPS-like planning paradigm (with fairly expressive operators) in which the rewritings yield complete and consistent plans, as opposed to complete but inconsistent variable assignments. In work on scheduling and rescheduling, (Monte Zweben & Deale 1994) define a set of general, but fixed, repairs methods, and use simulated annealing to search the space of schedules. Our plans are networks of actions as opposed to the metric-time total-order tasks in that work. Also we can easily specify different rewriting rules (general or specific) to suit each domain, as opposed to their fixed strategies. Related ideas have been used in heuristic search (Ratner & Pohl 1986). In that work, first they find a valid sequence of operators using an approximate algorithm. Then, they identify segments of this sequence, take their initial and end states, and heuristically search for a shorter path for that segment (the cost metric is the path length). They are doing a state-space search, while PBR is doing a plan-space search. The least-committed partial-order nature of PBR allows it to optimize the plans in ways that cannot be achieved by optimizing linear subsequences.

A variety of research has attacked the complexity of planning. Some systems incorporate automatically learned search control, for example, search control rules (Minton 1988) and abstraction (Knoblock 1994a). Our system does not learn the rewriting rules currently (see Future Work). Other work has reduced planning to propositional satisfiability, which can be solved by stochastic local search (Kautz & Selman 1996). These approaches do not specifically address plan quality, or else they consider only very simple cost metrics (such as the number of steps). Quality-improving control rules are learned in (Pérez 1996), but planning efficiency was not significantly improved. By exploiting domain-specific knowledge, conveniently expressed as plan rewriting rules, and the local search approach, we improve both plan efficiency and quality. Moreover, we provide an anytime algorithm while other approaches must run to completion.

Some domain specific planners have also used a transformational approach, for example, query evaluation in centralized databases (Graefe & DeWitt 1987). They parse the query to obtain an initial evaluation plan and iteratively transform this plan using a set of rules based on the algebra of the data model. PBR offers a more general and easily extensible framework to tackle more complex information gathering domains. Finally, the research in graph rewriting (Schurr 1996) may provide efficient matching algorithms and perhaps another implementation vehicle using high-level graph-rewriting programming languages.

Future Work

There are several issues that we plan to address more thoroughly in the future: initial plan generation, automatic rule generation, and alternative search strategies. Initial plan generation is domain-specific, but we intend to provide a domain independent procedural plan construction language to allow the convenient specification of plan construction algorithms. It will include primitives for adding steps and ordering constraints, but it will hide the complexity of the data structures used to represent the actual plans.

We believe that the rules can be generated by fully automated procedures in many domains. The methods can range from static analysis of the domain operators to analysis of sample equivalent plans (that achieve the same goals but at different costs). Note the similarity with methods to automatically deduce search control (Minton 1988; Etzioni 1993) and also the need to deal with the utility problem.

There are many techniques in the local search literature that we could adapt to our framework. In particular, we plan to explore variable depth rewriting, and variations of tabu search. In *variable depth*, a sequence of rewritings is applied in order to overcome initial cost increases that eventually would lead to strong cost reductions. This idea leads to the creation of rule programs, which specify how a set of rules are applied to the plan, possibly depending on run-time conditions. In *tabu search*, some of the rewritings are temporarily forbidden regardless of their cost. This is useful to avoid returning to some previously visited plan and thus cycling. Also, it forces the search not to be concentrated in a small local area around a local minimum. Finally, we plan to improve the planner implementation. For example, a RETE-like graph matcher (Forgy 1982) would make the system much more efficient.

Conclusions

We presented a new paradigm for efficient high-quality planning based on local search and plan rewriting, and we provided initial experimental support for its usefulness in several domains. This framework achieves a balance between domain knowledge, conveniently expressed as plan rewriting rules, and general local search techniques that have been proved useful in many hard combinatorial problems. We expect that these ideas will push the frontier of solvable problems for many domains into the range of real-world problems in which high quality plans and anytime behavior are needed.

Acknowledgments

The research reported here was supported in part by Rome Laboratory of the Air Force Systems Command and the Advanced Research Projects Agency under contract no. F30602-94-C-0210, and in part by the National Science Foundation under grant number IRI-9313993. The authors wish to thank the anonymous reviewers for their useful comments.

References

Arens, Y.; Knoblock, C. A.; and Shen, W.-M. 1996. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration* 6(2/3):99–130.

Dean, T., and Boddy, M. 1988. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 49–54.

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

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

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19:17–37.

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

Graefe, G., and DeWitt, D. J. 1987. The EXODUS optimizer generator. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* 16(3):160–172.

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

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*.

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

Knoblock, C. A. 1994b. Generating parallel execution plans with a partial-order planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*.

Knoblock, C. A. 1995. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.

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*.

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

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

Monte Zweben, B. D., and Deale, M. 1994. Scheduling and rescheduling with iterative repair. In *Intelligent Scheduling*. San Mateo, CA: Morgan Kaufman. 241–255.

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

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

Paulokat, J., and Wess, S. 1994. Planning for machining workpieces with a partial-order, nonlinear planner. In *Working notes of the AAAI Fall Symposium on Planning and Learning: On to Real Applications*.

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

Pérez, M. A. 1996. Representing and learning quality-improving search control knowledge. In *Proceedings of the Thirteenth International Conference on Machine Learning*.

Ratner, D., and Pohl, I. 1986. Joint and LPA*: Combination of approximation and search. In *Proceedings of the Fifth National Conference on Artificial Intelligence*.

Schurr, A. 1996. *Programmed Graph Replacement Systems*. World Scientific.

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

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