Cost-Based Learning for Planning

Srinivas Nedunuri and **William R. Cook**
Dept. of Computer Science, University of Texas at Austin
{nedunuri,wcook}@cs.utexas.edu

Douglas R. Smith
Kestrel Institute, Palo Alto
smith@kestrel.edu

Abstract

Most learning in planners to date has been focused on speedup learning. Recently the focus has been more on learning to improve plan quality. We introduce a different dimension: learning not just from failed plans, but learning from inefficient plans. We call this *cost-based learning* (CAL). CBL can be used to improve both plan quality and provide speedup learning. We show how cost-based learning can also be used to learn plan rewrite rules that can be used to rewrite an inefficient plan to an efficient one, in the style of Planning by Rewriting (PbR). We do this by making use of dominance relations. Additionally, the learned rules are compact and do not rely on state information so they are fast to match.

1 Introduction

One way to produce good quality plans is to transform the output of a fast but lower quality planner using plan rewriting (YFGG08; PMP$^+$03; AKM05). Plan rewriting was investigated quite extensively by Ambite et al. (AKM00; AKM05). They demonstrated impressive improvements in plan quality across a number of domains, even orders of magnitude in one (Distributed Query Optimization). Plan rewriting works by iteratively applying rewrite rules to an existing plan. One drawback of Ambite et al.'s particular approach is that some rules do nothing to improve plan quality, and can even lead to cycling (e.g. rules that do a simple transposition of two actions), so they must be applied carefully. Another more significant drawback is the need for a user to supply the rewrite rules, which is an error prone and time consuming task. In this paper we show how such rewrite rules can be automatically learned. Additionally, the learned rules are guaranteed to improve plan quality. Although Ambite et al. (AKM05) and others (eg. (NM10)) have looked at learning rewrite rules or plan improvement rules, the learned rules are often dependent on context or state in order to be applied, which makes them more expensive to apply and can lead to the utility problem that plagued early EBL approaches (Min90). Ambite et al.'s work is discussed further in the section on Related Work. The rewrite rules we learn do not depend on state or context so they are fast to match and apply. In order to do this we introduce a novel form of learning called *cost-based learning* (CBL) applied to search. CBL works by learning not just from planning failures (or successes) as conventional learning does but by learning from inefficient plans. We do this by applying *dominance relations* to the planning problem. A dominance relation is typically characterized by a predicate over pairs of partial plans. If a pair of partial plans p and p' satisfy the predicate then p' is guaranteed to lead to a worse solution than p, and can therefore be discarded from the search. We show how to learn such *dominance pairs*, and then show that under some fairly relaxed conditions it is possible to remove the common prefix of both partial plans, leaving a pair (q, q') which can immediately be turned into a plan-improving rewrite rule $q' \Rightarrow q$, useable in *any* planning problem in the same domain. Using this approach we are able to automatically learn most of the (hand written) rewrite rules of Ambite et al, as well as some additional ones that were missed by them. The dominance pairs can also be used as they are learned to speed up the current search. Unlike similar approaches using EBL (Min90), our stored knowledge does not depend on current state, complicating the matching. Our patterns are simple sequences of operators that can be efficiently matched.

2 Background

2.1 Problem Specification

The starting point is a statement of the problem to be solved. Formally, a *problem specification* is a 4-tuple $\langle D, R, o, c\rangle$, where D is the domain of input values, R is the range of result values, $o : D \times R \rightarrow Boolean$ is an *output* or *post condition* characterizing the relationship between valid inputs and valid outputs, and $c : D \times R \rightarrow Nat$ is a *cost function* that is being optimized. The operators o and c take the input as an argument because they need information supplied with the input. The intent is that a function $f : D \rightarrow R$ that solves the problem will take an input $x : D$ (a problem *instance*) and return a *solution* $z : R$ that satisfies o (making it a *feasible* solution) and minimizes c.

Example 1. Problem specification for sorting

$$\begin{array}{lll} D & \mapsto & [Nat] \\ R & \mapsto & [Nat] \\ o & \mapsto & \lambda(x : D, z : R) \,.\, asBag(x) = asBag(z) \\ & & \quad\quad \wedge \forall i < \|z\| - 1 .\, z_i \leq z_{i+1} \\ c & \mapsto & \lambda(x : D, z : R) \,.\, 1 \end{array}$$

In Eg. 1 the domain D and the range R are instantiated to be the type of lists of natural numbers. The symbol $\mapsto$ is

$$
\begin{array}{lll}
D & \mapsto & \{ops : OpTbl, type : TypeTbl, init : State, goal : State\} \\
 & & TypeTbl = Id \longmapsto Type \\
 & & OpTbl = OpId \longmapsto OpInfo \\
 & & OpInfo = \{params : [Id], pre : State, post : State\} \\
 & & State = [Id \longmapsto StateVal] \\
 & & StateVal = Boolean \mid Nat \mid Id \\
R & \mapsto & [Action] \\
 & & Action = \{opId : OpId, args : [Id]\} \\
o & \mapsto & \lambda(x,z)\,.\,\sigma(x.init, z) \supseteq x.goal \\
 & & \sigma(s, p{+}{+}[a]) = \textbf{let}\ acc = \sigma(s,p) \\
 & & \qquad\qquad aPre = (x.ops(a.opId).pre)\theta_a \\
 & & \qquad \textbf{in if}\ acc \supseteq aPre\ \textbf{then}\ \tau(acc,a)\ \textbf{else}\ \emptyset \\
 & & \sigma(s,[]) = s \\
 & & \tau(s,a) = \textbf{let}\ aPost = (x.ops(a.opId).post)\theta_a \\
 & & \qquad \textbf{in}\ s \ll aPost \\
 & & \theta_a = x.ops(a.opId).params \longmapsto a.args \\
c & \mapsto & \lambda(x,z)\,.\,\|z\|
\end{array}
$$

Figure 2.1: Specification for Planning

to be read as "translates to" and the "[,]" as "list of". The output condition o is a predicate, written as a lambda expression, that requires that the two arguments x of type D and z of type R when viewed as bags contain the same elements, and furthermore that every element of z except the last be smaller than its successor. This is not an optimization problem so the cost function c is constant. Any algorithm for sorting that meets this specification (such as quicksort, insertion sort, etc.) is considered correct.

Specification of (Classical) Planning Fig. 2.1 gives a problem specification for planning problems[1]. The reason for this particular specification format is that the development environment we use, called Specware (S), can check the specification for errors and also provide a customizable search program to implement the specification, as described in Section 2.2. The explanation of it (including notation) is as follows: The domain (type of the input to a planner) is collection of operators, a types table, an initial state, and a goal state, each of which has a type, analogous to a type in language like Java. For this reason, it is written as a record type$\{ops : Ops, type : TypeTbl, init : State, goal : State\}$, where $f : t$ means field f has type t. The *TypeTbl* is another structured type, in this case a *finite map* (written $Id \longmapsto Type$) which returns the PDDL type of an id (e.g. for Blocksworld, *type(Block-1)* would return *Block*). Similarly, *OpTbl* is another finite map, in this case returning the information (*OpInfo*) pertaining to a given operator id (such as *Stk* or *UnStk*). $OpInfo$ is a record type that gives the parameter list and pre and post conditions for each operator. Since *params* is a list of *Id*, its type is denoted *[Id]*. We use the state variable representation (GNT04) in which state is a list of state components (one for each property of interest), each of which is a finite map. Each entry in the map corresponds to a state variable (e.g. if *on* is a map then *on(A), on(B),* etc. are state variables). The output type (R) of the

[1]Translating from a standard format such as PDDL to this form is straightforward.

Op. Name	Params	Precond	Postcond
stk	a, t, c	$\{clr?(a), clr?(c)\}$ $\{on(a) = t\}$	$\{\neg clr?(c)\}$ $\{on(a) = c\}$
ust	a, b, t	$\{clr?(a), \neg clr?(b)\}$ $\{on(a) = b\}$	$\{clr?(b)\}$ $\{on(a) = t\}$
tr	a, b, c	$\{clr?(a), clr?(c)\}$ $\{on(a) = b\}$	$\{\neg clr?(c)\}$ $\{clr(b)\}$ $\{on(a) = c\}$

Table 1: Specification of the operators in Blocks World

planner is a sequence of actions. Each action is specified by an operator id and a list of arguments (meaning the corresponding operator is instantiated with those arguments). The output condition, o, is a boolean function (a λ term) requiring that the final state of the system (determined by the state function σ) is a superset of the goal state. The recursive call in σ determines the state just before the final action (if there is one) in a sequence of actions and checks that this state contains the precondition of the final action (ie. the final action is enabled) and if so, applies the state transition function τ to determine the next state. The $\ll$ operator in τ updates the state s with the postcondition of the action, leaving alone any terms that are not changed by the action postcondition (this ensures the frame axioms are satisfied). Evaluation of both σ and τ uses the substitution θ binding operation parameters to arguments. Finally, the cost of a plan is simply the length of the plan (but could in general be any compositional cost function). At this point, we have a specification of Planning *in genera*l. A particular planning domain is then an *instance* of this specification, as the next example demonstrates.

Example 2. Blocks World (BW)

To create a planner to solve Blocks World, the ops field of the input x contains the operator map shown in Table 1, containing three operators: *stk*, which stacks a block from the table onto another block, *ust*, which unstacks a block onto the table, and *tr*, which transfers a block from one supporting block to another. State is represented with the two finite maps, $clr? : Id \longmapsto Boolean$ and $on : Id \longmapsto Id$. An empty map means that particular state component is unspecified. The *types* table gives the types of all the domain objects as well as the parameters to the operations. For BW, a, b, c have the type Blk, t has the type Tbl Finally, we specify a particular BW instance. For example, an initial state of three blocks A, B, C (all with type Blk) all on the table T (of type Tbl) is represented by $x.init = \{\{clr?(A), clr?(B), clr?(C)\}, \{on(A) = T, on(B) = T, on(C) = T\}\}$ and a goal of A on B on C is written $x.goal = \{on(A) = B, on(B) = C, on(C) = T\}$. Notice that the input x combines both the BW domain description as well as a particular instance of the BW problem. Another way of viewing it is as a two stage process: instantiate the ops field in the input to get a planning domain, and then instantiate the *init* and *goal* to get a planning instance.

Algorithm 1 Program Schema for Global Search

```
def start(x:D):[R]×DomR = search(x,[],initSpace(x))

def search(x:D, best_so_far:[R], y:R̂):[R]×DomR =
    if not (filter(x,y) then (best_so_far,[])
    else let dom_pair=testForDominance(x,best_so_far,y) in
        if dom_pair /= null_pair
        then (best_so_far,[dom_pair])
        else let
            soln = extract(y)
            best_now = opt(best_so_far ++ soln)
            (childrens_best,dom_reln) =
                searchCh x best_now y (subspaces(x,y))
            new_best = opt(best_now ++ childrens_best)
            in (new_best,dom_reln)

def subspaces(x:D,y:R̂) = [y′: split(x,y,y′)]

def searchCh(x:D,best_so_far:[R],chldrn:[R̂]):[R]×DomR =
    //foldl is a higher order function that ‘‘updates’’
    //the initial pair (best_so_far,[]) with result of
    //searching each y∈ys using (seeIfTheresBettrSoln x)
    foldl (seeIfTheresBettrSoln x)(best_so_far,[]) chldrn

def seeIfTheresBettrSoln:D -> (accum:(R×DomR)):R×DomR =
    let (best_so_far,dom_reln)= accum
        (p’s_best,p’s_dom_reln)= search(x,best_so_far,p)
    in (opt(best_so_far++p’s_best), dom_reln++p’s_dom_reln)
```

One valid output or plan z would be the list of actions $[stk(B,T,C), stk(A,T,B)]$. It is straightforward to verify that this constitutes a valid plan by confirming it satisfies the definition of o after expanding the function definitions. The cost of this plan is 2. Another valid plan, with a cost of 3, is $[stk(B,T,A)$,$tr(B,A,C)$,$stk(A,T,B)]$. The search program for constructing these plans is described next.

2.2 Global Search

Global Search (GS) (Smi88) (also called *Abstract Search* or *Refinement Search* (GNT04)) provides one approach to computing a solution to a problem specification by recursive decomposition of a *search space*, using the operations of branching, pruning, and solution extraction. Since spaces can be quite large, even infinite, they are not represented extensionally but intensionally, through a descriptor of some sort. However to avoid being pedantic, the term space is used instead of space descriptor.

A *schema* (akin to a template function in Java) for GS is shown in Alg. 1, written in the executable subset of *MetaSlang*, a specification language in the *Specware* development environment (S). The executable sublanguage is a pure higher order functional language in the style of Haskell[2]. That is, all functions are defined in terms of other functions, including recursive calls to the function being defined. There are no side-effecting assignment statements as there are in a language like Java. A backend code generator generates code in one of a number of different target languages, including Lisp, Java, and Haskell. However, no familiarity with MetaSlang is assumed and English language descriptions of all code are provided, which we now do.

The declaration of *search* says that it takes an argument *x* of type *D*, the best solution so far (represented as a list), and the current space to search, y, of type $\widehat{R}$ and returns a pair, consisting of a list of solutions, of type *[R]*, and a dominance relation of type *DomR* (explained later). Search first passes the space through a **filter**. A filter is a predicate which is some relaxed form of the output condition, o, that is easy to evaluate . If the space passes the filter, then if the *testForDominance* passes (explained in Section 3.2), the search attempts to **extract** a solution, and determines whether the best solution so far or the extracted solution is the better one. The better one along with the list of subspaces of the current space are passed on to *searchCh* which recursively searches each child, returning the best solution it finds. Finally, that is compared with the better one, and the best returned. The search is initiated by the function *start* which because it has no solutions yet simply passes an empty list and a descriptor returned by the **initSpace** function, corresponding to the space of all possible solutions. Because solutions are extracted from spaces, a space is also called a *partial solution* or sometimes a *node* in a search tree. To use the schema, the type $\widehat{R}$ and the operators **initSpace, extract, filter**, and **split** need to be instantiated.

Type and Operator instantiation for Planning Partial plans have just the same structure as complete plans, namely a list of actions, so the type $\widehat{R}$ is the same as R. For this reason, when there is no confusion, references to plans also apply to partial plans. The *initSpace* operator just returns an empty list. The *split* operator appends some action (chosen from all the possible actions, that is all possible instantiations of operators by assignment of type-compatible domain objects to parameters) to the partial plan. *Filter* ensures that the appended action is enabled by the preceding partial plan. *Extract* can extract a complete plan at any time (it may of course be infeasible). Specware automatically composes the program schema with the instantiations to produce an executable program.

2.3 Dominance Relations

If a pair of spaces is in a dominance relation, the first will always lead to at least as “good” an optimal solution as the second, where “goodness” is measured by some cost function on solutions. The first one is said to *dominate* the second, which can be eliminated from the search. Dominance relations have a long history in search (Iba77). Here though we follow the approach of Nedunuri and Cook (NC09) which is briefly summarized below. For readability, ternary relations that take the input (x) as one of their arguments are shown in subscripted infix form and implicitly quantified over (eg. $\forall x.\ \rhd(x,a,b)$ is written $a \rhd_x b$). $\oplus$ denotes a left-associative domain specific operator used to *extend* a partial solution. That is $y \oplus e$, obtained by extending the partial solution y with e (called an *extension*), denotes a new partial solution that is more defined than y (ie. if a solution can be derived from $y \oplus e$ then it can be derived from y). Its definition depends on $\widehat{R}$ and the type of

[2] Unlike Haskell, MetaSlang is strict.

e (e.g. if $\widehat{R}$ is a list type and e is a list, then $\oplus$ might be list concatenation, $++$). A cost function c is *compositional* if $c(x, u \oplus v) = c(x, u) + c(x, v)$.

Definition 1. *Semi-Congruence* is a relation $\rightsquigarrow_x \subseteq D \times \widehat{R}^2$ such that

$$\forall e, y, y' \,.\, y \rightsquigarrow_x y' \Rightarrow o(x, y' \oplus e) \Rightarrow o(x, y \oplus e)$$

That is, semi-congruence ensures that any feasible extension of y' is also a feasible extension of y.

Definition 2. *SC-Dominance* is a relation $\widehat{\rhd}_x \subseteq D \times \widehat{R}^2$ such that

$$\forall e, y, y' \,.\, y \widehat{\rhd}_x y' \Rightarrow$$
$$o(x, y \oplus e) \wedge o(x, y' \oplus e) \Rightarrow c(x, y \oplus e) < c(x, y' \oplus e)$$

That is, sc-dominance ensures that one feasible completion of a partial solution is less expensive[3] than the same feasible completion of another partial solution. The following theorem and proposition show how the two concepts are combined.

Theorem 1. *If $\rightsquigarrow_x$ is a semi-congruence relation, and $\widehat{\rhd}_x$ is a sc-dominance relation, and $c^* : \widehat{R} \rightarrow Nat$ denotes the least cost solution in a space, then*

$$\forall y, y' \,.\, y \widehat{\rhd}_x y' \wedge y \rightsquigarrow_x y' \Rightarrow c^*(y) < c^*(y')$$

When $y \widehat{\rhd}_x y' \wedge y \rightsquigarrow_x y'$ we say y *dominates* y', written $y \rhd_x y'$. The collection of pairs (y, y') such that y dominates y' forms the *extension* of the dominance relation.

The following proposition shows how to get a straightforward sc-dominance condition. Note that we have lifted the cost function to partial solutions.

Proposition 1. *If c is compositional then $c(x, y) < c(x, y')$ is a sc-dominance relation*

For Planning, the $\oplus$ operator is simply list concatenation, denoted $++$.

3 Learning Rewrite Rules

We now describe the contribution of this paper which is twofold. First we define a domain-independent dominance relation which is applicable to all planning problems. Given such a definition, and the instantiated program schema of Alg. 1, any two nodes p, p' in the search tree can be tested at run-time to see if one dominates the other. In general, performing this test on all pairs of nodes in a search tree is computationally infeasible, but we only need small examples to discover useful dominance pairs, so its cost is acceptable. The second part of our contribution is to show how to generalize such pairs and then extract a pair of context-free plan *segments* q, q'. The pair (q, q') forms a rewrite rule which can now be applied to any plan in the domain to get an improved plan, for example one generated by a custom planner. Furthermore, the rewrite rules can be applied to the dominance pairs themselves to simplify them relative to each other. In this way, the large number of learned dominance pairs often reduces to a handful of small useful rules.

[3] More generally, it is sufficient if $c(x, \widehat{z} \oplus e) \leq c(x, \widehat{z}' \oplus e)$ but we are looking for a guaranteed improvement, so we use the strict inequality

3.1 A Dominance Relation for Planning

First we derive a semi-congruence condition, which (Def. 1) ensures that if one partial plan p' can be feasibly extended with an extension, then so can another plan p with the same extension. That is, we seek a condition between p and p' that ensures $\forall e.\, o(x, p' \oplus e) \Rightarrow o(x, p \oplus e)$. We find this by backwards calculation from the conclusion. Before doing so, we need the following proposition which provides a way of calculating the state σ after extending a partial plan with a given extension

Proposition 2. $\forall s, p, e \,.\, \sigma(s, p{+\!+}e) = \sigma(\sigma(s, p), e)$

The calculation of the required condition is:

$$\begin{array}{ll} & o(x, p \oplus e) \\ = & \{\text{defn of } o\} \\ & \sigma(x.init, p{+\!+}e) \supseteq x.goal \\ = & \{\text{Prop. 2}\} \\ & \sigma(\sigma(x.init, p), e) \supseteq x.goal \\ \Leftarrow & \{o(p')\, ie.\, \sigma(\sigma(x.init, p'), e) \supseteq x.goal\} \\ & \sigma(x.init, p) \supseteq \sigma(x.init, p') \end{array}$$

That is, p is semi-congruent with p' if the state after executing partial plan p from an initial state is a superstate of the state of partial plan p' executed from the same initial state. Combining this with Thm 1 we conclude that p dominates p' if $\sigma(x.init, p) \supseteq \sigma(x.init, p') \wedge c(x, p) \leq c(x, p')$.

3.2 Learning Ground Dominance Pairs

To learn a dominance pair, suppose the search has previously explored one path, finding a solution z. Now suppose the search reaches a current partial solution p'. If some ancestor p of z dominates p' then p' need not be searched any further and the pair (p, p') is added to the extension of the dominance relation. This idea is implemented in the *testForDominance* procedure in Alg. 1 which returns the pair (p, p') if p dominates p' and the null pair otherwise. The term *dom_reln* contains the current set of such pairs, which is returned to the top level when the search completes.

Example 3. Blocks World

Consider the BW input in Ex.3. Suppose the search has already discovered the solution $z = [stk(B, T, C), stk(A, T, D), stk(A, D, B)]$ and is currently at the partial solution $y''_1 = [stk(B, T, A)]$. No ancestor of z is semi-congruent with this partial solution. The same holds for y''_2. The search continues to $y''_3 = [stk(B, T, A), ust(B, A, T), stk(B, T, C)]$ with which the ancestor y_1 of z is (the highest ancestor which is) semi-congruent. y_1 is also cheaper than y''_3 and so no plan that follows from y''_3 will be better than z. Therefore the pair (y_1, y''_3) can be added to the dominance relation.

3.3 Generalization to First Order Dominance Pairs

The resulting set of dominance pairs could be considered the ground extension of a domain-specific dominance relation. The first step is to parameterize it to a first order (but still extensional) relation. This can be done using either the EGGS generalization mechanism of Mooney and

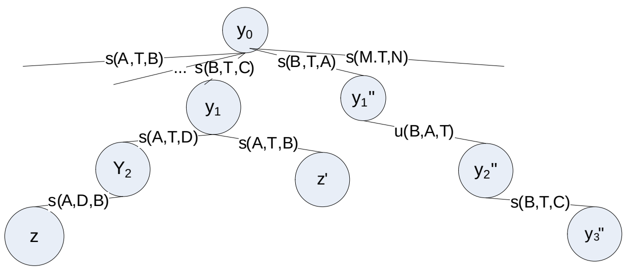

Figure 3.1: Dominance example for Blocks World (only the relevant portion of the search tree is shown)

Bennett (MB86) or the mechanism of Kambhampati et al (KKQ96). Kambhampati's approach is the more straightforward one: it allows for the replacement of any constants by variables provided the domain theory does not refer to any object constants by name (for example if the specification of the stk operator referred to the table T in either the pre or post condition, it would not be a name insensitive theory[4]. For example, generalization of the dominance pair in Eg. 3 is $\forall a, b, c : Blk, t : Tbl. [stk(b, t, c)]) \rhd_x [stk(b, t, a), ust(b, a, t), stk(b, t, c)]$. This dominance pair can be used elsewhere in the search to prune off unpromising spaces by skipping branches that match the second element of the dominance pair.

3.4 Generalization to Rewrite Rules

The second step is to try to generalize a dominance pair in the relation to one applicable to any blocks world problem instance. This requires identifying those pairs of plan segments that do not depend on the initial state. For example, $[ust(b, c, t), stk(b, t, c)]$ is a useless series of steps no matter what the common prefix is and can always be replaced with the empty sequence []. That is $[] \rhd_x [ust(b, c, t), stk(b, t, c)]$.

Under what circumstances can the common prefix be stripped off a dominance pair? Intuitively, it is when the dominated branch relies on what is established by the prefix (to achieve its current state) at least as much as the dominating branch does. This can be determined by *regressing* a state (in the manner described in (KKQ96)) back up the tree. Regressing a state over a series of branches simply amounts to computing the weakest precondition of the given series of branches. It determines what state must hold before the series of branches in order to ensure the given state at the end. Its formal definition is as follows:

Definition 3. The *regression* of a state s over an extension e denoted $\sigma^{-1}(s, e)$, is defined as:

$$\begin{aligned} \sigma^{-1}(s, e \oplus b) &= \sigma^{-1}(\sigma_p^{-1}(s, b), e) \\ \sigma^{-1}(s, \varepsilon) &= s \end{aligned}$$

[4]However, it is easy to turn it into one: just replace the constant T in the pre/post conditions with a variable t, define a type Tbl (or equivalently a predicate such as *tbl*?) and assert that t's type is Tbl. The problem input would specify that T is a table by asserting its type is Tbl. This is what we have done.

where b is the branch to the partial solution from its parent, ie.$split(x, e, e \oplus b)$. $\sigma_p^{-1}(s, b)$ is a primitive regression step whose definition in the case of planning is $\sigma_p^{-1}(s, a) = (s - a.post) \cup a.pre$.

Definition 4. The *smallest prestate* of a non-empty plan $e \oplus b$ denoted $\sigma^{-1}(e \oplus b)$ is defined as $\sigma^{-1}(b.pre, e)$.

The smallest prestate (*sp*) of a plan gives the smallest state that must hold at the start of the plan to ensure the final action in the plan is successfully executed. Finally, let $W(p)$ be set of state variables whose values are modified by plan p (that is, their values at the end of executing p are different from the their values at the start of p). The following theorem defines when it is safe to strip off the common prefix:

Theorem 2. Given a compositional cost function, for all x, q, q' :

$$(\exists p. \, p \oplus q \rhd_x p \oplus q') \wedge \sigma^{-1}(q) \subseteq \sigma^{-1}(q') \wedge W(q) = W(q') \Rightarrow \forall p' : \widehat{R}. \, p' \oplus q \rhd_x p' \oplus q'$$

Intuitively, the theorem says that if some partial plan $p \oplus q$ dominates another partial plan $p \oplus q'$ and the *sp* of q is no bigger than that of q', and both q and q' modify the same state variables, then for *any* p', $p \oplus q$ dominates $p \oplus q$. Finally, the following theorem states that it is profitable to carry out such a rewrite on any feasible plan π

Theorem 3. $\forall q, q'. \, o(x, \pi) \wedge (\forall p'. \, p' \oplus q \rhd_x p' \oplus q') \Rightarrow c(x, \pi[q' := q]) < c(x, \pi)$

Example 4. Blocks World. Returning to Fig. 3.1, suppose the (generalized) solution $z' = [stk(b, t, c), stk(a, t, b)]$ is discovered first and then the (generalized) solution $z = [stk(b, t, c), stk(a, t, d), tr(a, d, b)]$. The extension of z' from the lowest common ancestor of z' and z, namely y_1, is $[stk(a, t, b)]$. The smallest prestate $\sigma^{-1}([stk(a, t, b)])$ is $\sigma^{-1}(\{clr?(a), clr?(b), on(a) = t\}, []) = \{clr?(a), clr?(b), on(a) = t\}$. For z, its extension from the ancestor y_1 is $[stk(a, t, d), tr(a, d, b)]$ and its smallest prestate calculated in a similar manner is giving $\{clr?(a), clr?(b), clr?(d), on(a) = t\}$, which is a superset of $\{clr?(a), clr?(b), on(a) = t\}$. Finally, $W([stk(a, t, b)]) = W([stk(a, t, d), tr(a, d, b)]) = \{on(a), clr?(b)\}$. Therefore the sequence $[stk(a, t, d), stk(a, d, b)]$ can be replaced with $[stk(a, t, b)]$ in *any* BW plan.

3.5 Efficiency Considerations

For efficiency reasons, we do not attempt to match a partial solution with every previously discovered partial solution, but only with the current best solution. Also, the regression is done incrementally as the search tree is unwound, and is cached for the currently best known solution.

4 Experiments

We ran our learning algorithm on a number of domains taken from (AKM05) as well as the one from the 3rd International Planning Competition (IPC). Some sample results are described below.

4.1 Blocks World

Given a simple input of 3 blocks, the learning system learnt both the (manually written) rules in (AKM05) shown below

$$[stk(a,t,c), ust(a,c,t)] \Rightarrow []$$
$$[ust(a,b,t), stk(a,t,c)] \Rightarrow [stk(a,b,c)]$$

Using these rules, Ambite et al. were able to achieve an average reduction in plan length over a naive plan of about 20%. The naive plan was generated by a custom planner that first unstacked all the blocks to the table, and then stacked them. This avoids having to ever having to move a block directly from one block to another. In addition, our learning system learned an additional rule, $[stk(a,t,b), tr(a,b,c)] \Rightarrow [stk(a,t,c)]$ but the left hand side does not occur in the naive plan so it is not used.

4.2 Logistics

The Logistics problem consists of delivering each of a number of packages from its current location to the desired location using a truck. The operators in the domain are *l(oad)*, *u(nload)*, and *d(rive)*. Given a simple input of 2 packages and 2 locations, the planner learns the *Loop* rule of Ambite ($[d(t,a,b), d(t,b,a)] \Rightarrow []$) as well as a rule not mentioned by them: ($[u(p,t,a), l(p,t,a)] \Rightarrow []$). Given an input with 3 packages and 3 locations, the planner learns their *Triangle Inequality* rule: ($[d(t,a,b), d(t,b,c)] \Rightarrow [d(t,a,c)]$). They also have another rule (*Load Earlier*) which their rule learning algorithm is unable to learn. *Load Earlier* suggests loading a package at the earliest opportunity to save having to potentially make a specific trip later to pick up that package. The extra trip can occur any number of steps later. Because we learn specific sequences, our learning system is unable to learn the most general form of this rule, but learns instead the specific cases where the extra trip occurs 1,2,3... steps later. For example, the 1 step form of the rule it learns is $[d(t,a,b), l(p,t,b), d(t,b,a), l(q,t,a), d(t,a,b)] \Rightarrow [l(q,t,a), d(t,a,b), l(p,t,b)]$. Using the *Loop*, *Triangle Inequality*, and *Load Earlier/Unload Later* rules, Ambite et al. were able to achieve an average reduction in plan length from a naive plan of over 40%.

4.3 ZenoTravel (3rd IPC)

The domain definition translated from the Strips PDDL description is shown in Table 2. State is represented with three finite maps, *at*, giving the location of a person or airplane, *fl*, giving the current fuel level of the airplane, and *dec*, which is a table of consecutively decreasing fuel levels (*dec* ensures that there is enough fuel for the flight) Given a simple input with 2 people, and 2 cities, and 1 plane, the learning system learns several hundred dominance pairs (rules). After using smaller rules to simplify larger rules, they reduce down to a handful of rules, of which some of the interesting left-hand sides are: (all rewrite to the empty list []) $[em(p,a,c), dem(p,a,c)]$, $[ref(a,f,l,m), fly(f,t,m,l), fly(t,f,l,k), ref(a,f,k,l)]$, and $[ref(a,f,k,l), fly(f,t,l,k), ref(a,t,k,l), fly(t,f,l,k)]$. Given 3 cities, it also learns $[ref(a,c,k,l), fly(f,d,l,k), ref(a,d,k,l), fly(d,e,l,k)] \Rightarrow [ref(a,c,k,l), fly(c,e,l,k)]$. Applying these rules to naive plans resulted in an average plan length reduction of around 25%. The naive planner visits each city in turn, picking up all the passengers, and taking each one in turn to their destination.

Op.Name	Params	Precond.	Postcond.
em	p,a,c	$\{at(p)=c,$ $at(a)=c\}$ $\{\}$ $\{\}$	$\{at(p)=a\}$ $\{\}$ $\{\}$
dem	p,a,c	$\{at(p)=a,$ $at(a)=c\}$ $\{\}$ $\{\}$	$\{at(p)=c\}$ $\{\}$ $\{\}$
fly	a,f,t,l,k	$\{at(a)=f\}$ $\{fl(a)=l\}$ $\{dec(l)=k\}$	$\{at(a)=t\}$ $\{fl(a)=k\}$ $\{\}$
zoom	a,f,t,l,k,j	$\{at(a)=f\}$ $\{fl(a)=l\}$ $\{dec(l)=k,$ $dec(k)=j\}$	$\{at(a)=t\}$ $\{fl(a)=j\}$ $\{\}$
ref	a,f,k,l	$\{at(a)=f\}$ $\{fl(a)=k\}$ $\{dec(l)=k\}$	$\{\}$ $\{fl(a)=l\}$ $\{\}$

Table 2: Specification of the operators for Zeno Travel

Input Size (n)	Naive Plan Length	Rewritten Plan Length	FF Plan Length	FF Time Taken
10	76	54	36	0s
20	179	127	80	0s
40	290	218	138	1s
80	588	448	308	41s
160	1220	920	-	>30 m

Table 3: Comparison of Plan Length and Times with FF

Table 3 compares the output of our naive planner along with with the rewritten plan obtained by applying the learned rewrite rules to the naive plan with the results of running FF (HN01), a state of the art planner, on the same inputs. For simplicity we consider n passengers in n cities and 1 plane. *In all cases, the total time taken by our naive planner plus the rewrite engine was under a second.* In contrast, the time taken by FF appears to grow exponentially. Although the resulting plan length was about 50% longer than what was produced by FF[5], our system scales much better as Tbl 3 shows. We also tried the more recent Fast Downward planner (Hel06) with a variety of heuristics (landmark-cut, merge-and-shrink, and blind) but the planning times were longer than they were for FF.

5 Summary and Further Work

Currently a custom hand-written planner is used to produce an initial plan. More work is needed to integrate learned in-

[5] We are currently working on synthesizing domain-specific planners which will reduce this difference considerably

formation into state-of-the-art domain independent heuristic planners such as FF (HN01) and FD (Hel06). As an alternative, we are working on *synthesizing* domain specific satisficing planners, continuing the early work of Srivastava and Khambampati (SK98). Such planners are synthesized by the use of *domain-specific* dominance relations, with the intent of reducing the branching in the search space, sometimes at the cost of extra plan length. The rewrite rules are then applied the same way as they are now to the output of such planners to produce a near-optimal plan.

We also do not currently handle constraints or temporal planning. We expect to address both limitations in future work.

6 Related Work

Dominance relations appear not to have been used much in planning. A rare exception is Mills-tettey et al (MtSD06) who incorporate a form of dominance into a regression path planner with good results. Yu and Wa (YW88) study how to inductively learn intentional definitions of dominance relations. They demonstrate their approach on a variety of knapsack problems and show good results. However, their learned rules are not logically sound.

Ambite et. al (AKM05) have investigated learning plan rewrite rules in great detail. They do this by comparing an initial inefficient plan with a plan generated by some other approach (e.g. local search). By doing a graph comparison they extract the rewrite rules. We will refer to their approach as Learning by Graph Matching (LGM). Our learning approach has the following advantages over LGM:

- LGM requires 2 complete plans to compare. Moreover, one of the plans has to be an optimal plan. We do not require complete or optimal plans (although in the interest of efficiency we often delay dominance testing until a complete plan has been found).
- LGM is a separate phase from planning. In our case, the learning mechanism could potentially be incorporated into the planner to speed up its current search.
- LGM relies on an approximation to testing for subgraph isomorphism. As such it misses some rewrites (such as the "Load Earlier" rule mentioned previously) that we are able to find (although our learned rule suffers from a different shortcoming, described earlier). The learned rules in LGM are also context-dependent, complicating the subsequent rewrite phase.
- LGM learns rules which do not by themselves improve plan quality (for example, simple interchanges of actions). Our learned rules are guaranteed to improve plan quality (for the given cost function c).

Our rewriting engine is also much simpler than theirs. We only need to match context-free sequences of actions, not context-dependent subgraphs. On the other other hand their use of partial order planning allows them to match subplans in which an action can precede another by an arbitrary number of actions. We cannot do that.

Using an earlier version of the Specware framework (KIDS), Srivastava and Khambampati (SK98) were able to successfully synthesize efficient domain-specific planners for several domains. However, they limited their attention to satisficing planners and did not attempt learning or consider dominance relations. We are able to automatically learn some of their pruning rules, eg. the "Limit Useless Moves" rule in BlocksWorld that avoids two consecutive moves of a block, and their rule in Logisitics that says planes should not make consecutive flights without loading or unloading a package.

EBL also generalizes explanations of failure, but early attempts ran into the Utility problem (Min90) on account of the large amount of learned information as well as the costs associated with matching. Although our current rewrite engine is extremely naive, it ought to be possible to make it much more efficient by a compact representation of the patterns coupled with efficient pattern matching algorithms such as Aho-Corasick or Rabin-Karp (CLRS01) along the lines of what is done in spell-checkers for large documents.

References

J.L. Ambite, C.A. Knoblock, and S. Minton. Learning plan rewriting rules. In *Artificial Intelligence Planning Systems (AIPS)*, 2000.

J.L. Ambite, C.A. Knoblock, and S. Minton. Plan optimization by plan rewriting. In *Intelligent Techniques for Planning*. 2005.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

M. Helmert. The fast downward planning system. *J. of AI Research*, 26:191–246, 2006.

J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *J. of AI Research*, 14, 2001.

T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24(2):264–279, 1977.

S. Kambhampati, S. Katukam, and Y. Qu. Failure driven dynamic search control for partial order planners: An explanation based approach. *Artificial Intelligence*, 88:253–315, 1996.

R. Mooney and S.W. Bennett. A domain independent explanation-based generalizer. In *AAAI-86*, 1986.

S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artif. Intell.*, 42, March 1990.

G. A. Mills-tettey, A. Stentz, and M. B. Dias. Dd* lite: Efficient incremental search with state dominance. 2006.

S. Nedunuri and W.R. Cook. Synthesis of fast programs for maximum segment sum problems. In *Intl. Conf. on Generative Programming and Component Engineering (GPCE)*, Oct. 2009.

H. Nakhost and M. Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *ICAPS*, pages 121–128, 2010.

J. Pineau, M. Montemerlo, M. Pollack, N. Roy, and S. Thrun. Towards robotic assistants in nursing homes:

Challenges and results. *Robotics and Autonomous Systems*, 42, 2003.

Specware. http://www.specware.org.

B. Srivastava and S. Kambhampati. Synthesizing customized planners from specifications. *J. of AI Research*, 8:61–75, 1998.

D. R. Smith. Structure and design of global search algorithms. Tech. Rep. Kes.U.87.12, Kestrel Institute, 1988.

S. Yoon, A. Fern, R. Givan, and C. Guestrin. Learning control knowledge for forward search planning. *J. of Mach. Learn. Research*, 9, 2008.

C-F. Yu and B. W. Wah. Learning dominance relations in combined search problems. *IEEE Trans. Softw. Eng.*, 14:1155–1175, August 1988.

7 Appendix: Proofs of Theorems

Proposition 2. $\forall s, p, e \,.\, \sigma(s, p{+\!+}e) = \sigma(\sigma(s, p), e)$

Proof. By induction. (In all cases where the definition of σ is expanded, we assume the non-empty branch, ie. the subsequent action is enabled. The empty branch is easy to demonstrate)

Base Case: $e = [a]$

$$
\begin{array}{ll}
 & \sigma(s, p{+\!+}[a]) \\
= & \{\text{unfold defn of } \sigma \text{ and } e\} \\
 & \tau(\sigma(s, p), a) \\
= & \{\text{let } p = fp{+\!+}[lp], p = [] \text{ case is trivial}\} \\
 & \tau(\tau(\sigma(s, fp), lp), a) \\
= & \{\text{intro } \sigma \text{ by folding base case of } \sigma\} \\
 & \tau(\sigma(\tau(\sigma(s, fp), lp), []), a) \\
= & \{\text{fold inductive case in defn of } \sigma\} \\
 & (\sigma(\tau(\sigma(s, fp), lp), [a]) \\
= & \{\text{replace } \tau \text{ by folding } \sigma\} \\
 & \sigma(\sigma(s, fp{+\!+}[lp]), [a]) \\
= & \{\text{fold } p = fp{+\!+}[lp], e = [a]\} \\
 & \sigma(\sigma(s, p), e)
\end{array}
$$

Inductive Case: Assume result holds for e, consider $e{+\!+}[a]$.

$$
\begin{array}{ll}
 & \sigma(s, p{+\!+}(e{+\!+}[a])) \\
= & \{\text{assoc. of } {+\!+}, \text{ defn of } \sigma\} \\
 & \tau(\sigma(s, p{+\!+}e), a) \\
= & \{\text{IH}\} \\
 & \tau(\sigma(\sigma(s, p), e), a) \\
= & \{\text{fold defn of } \sigma\} \\
 & \sigma(\sigma(s, p), e{+\!+}[a])
\end{array}
$$

□

Theorem 2. Given a compositional cost function, for all x, q, q' :

$$
(\exists p.\, p \oplus q \rhd_x p \oplus q') \wedge \sigma^{-1}(q) \subseteq \sigma^{-1}(q') \wedge W(q) = W(q') \Rightarrow \forall p' : \widehat{R}.\, p' \oplus q \rhd_x p' \oplus q'
$$

Proof. Let s and s' denote $\sigma(x.init, p' \oplus q)$ and $\sigma(x.init, p' \oplus q')$ resp. To demonstrate dominance we need to show that $s \supseteq s' \wedge c(x, p' \oplus q) \le c(x, p' \oplus q')$. Because the cost function is compositional, the SC-dominance condition follows by Prop. 1 so we focus on demonstrating semi-congruence, $s \supseteq s'$. Now given an assignment $v = a$ in s', there are two cases to consider: either $v \notin W(q')$ or $v \in W(q')$.

Case $v \notin W(q')$: Since it was not modified by q', the state variable v had the value a at the start of q'. (This follows from the definition of σ). By Proposition 2 $s = \sigma(\sigma(x.init, p'), q)$ and $s' = \sigma(\sigma(x.init, p'), q')$. Therefore any state assignment is present at the start of q iff it is also present at the start of q'. Assume $\sigma(x.init, p) \supseteq \sigma^{-1}(q')$, otherwise q' will not lead to a feasible plan. Now by the assumption $\sigma^{-1}(q) \subseteq \sigma^{-1}(q')$, and Lemma 1, any assignment at the start of q is present at the end of q unless overwritten . From the assumption $W(q) = W(q')$, v is also not in $W(q)$, ie. it is not modified by q. Therefore $v = a$ must also be present in $\sigma(x.init, p' \oplus q)$.

Case $v \in W(q')$: If, on the other hand, $v \in W(q')$ then again from the assumption $W(q) = W(q')$, it must be in $W(q)$. Suppose q last assigns b to v, ie. $v = b$ is present in σ. Then b must equal a otherwise we would not have $\sigma(x.init, p \oplus q) \supseteq \sigma(x.init, p \oplus q')$ as implied by the assumption $p \oplus q \rhd_x p \oplus q'$. □

Lemma 1. $s \supseteq \sigma^{-1}(p) \Rightarrow \forall (v = e) \in s.\, v \notin W(p) \Rightarrow (v = e) \in \sigma(s, p)$

Proof. By induction. For a single action plan $p = [a]$, given the antecedent and the definition of σ, the state is at least $aPre$, so by the definition of τ, the post state is $\sigma(s, [a]) \ll aPost$ but since a does not write v, $(v = e) \in \sigma(s, [a])$ as required. Assume now the result holds for a plan p and consider $p{+\!+}[a]$. If $v \notin W(p{+\!+}[a])$ then also $v \notin W(p)$ and by the IH, $(v = e) \in \sigma(s, p)$. Since $s \supseteq \sigma^{-1}(p{+\!+}[a])$, the required state $aPre$ is exceeded, so again from the definition of τ, the post state is $\sigma(s, p) \ll aPost$ but since a does not write v, $(v = e) \in \sigma(s, p{+\!+}[a])$ as required. □

Theorem 3. $\forall q, q' .\, o(x, \pi) \wedge (\forall p' .\, p' \oplus q \rhd_x p' \oplus q') \Rightarrow c(x, \pi[q' := q]) < c(x, \pi)$

Proof. Since $p' \oplus q \rhd_x p' \oplus q'$ for any p', it follows that $p \oplus q \rhd_x p \oplus q$, and from the definition of dominance that $p \oplus q \widehat{\rhd}_x p \oplus q$. Therefore $c(x, p \oplus q \oplus r) < c(x, p \oplus q' \oplus r)$ as required. □