Learning Action Strategies for Planning Domains

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:24947997">http://nrs.harvard.edu/urn-3:HUL.InstRepos:24947997</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Learning Action Strategies for Planning Domains

Roni Khardon

TR-09-97

May 1997

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts
Learning Action Strategies
for Planning Domains

Roni Khardon*
Aiken Computation Laboratory,
Harvard University,
Cambridge, MA 02138
roni@deas.harvard.edu

May 19, 1997

Abstract

This paper reports on experiments where techniques of supervised machine learning are applied to the problem of planning. The input to the learning algorithm is composed of a description of a planning domain, planning problems in this domain, and solutions for them. The output is an efficient algorithm — a strategy — for solving problems in that domain. We test the strategy on an independent set of planning problems from the same domain, so that success is measured by its ability to solve complete problems.

A system, L2Act, has been developed in order to perform these experiments.

We have experimented with the blocks world domain, and the logistics domain, using strategies in the form of a generalization of decision lists, where the rules on the list are existentially quantified first order expressions. The learning algorithm is a variant of Rivest's [39] algorithm, improved with several techniques that reduce its time complexity. As the experiments demonstrate, generalization is achieved so that large unseen problems can be solved by the learned strategies. The learned strategies are efficient and are shown to find solutions of high quality. We also discuss preliminary experiments with linear threshold algorithms for these problems.

1 Introduction

The problem of enabling an artificial agent to choose its actions in the world so as to achieve some goals has been extensively studied in AI, and largely in the subfield of planning. The general setup in planning assumes that agent has a model of the dynamics of each action, that is, when it is applicable and what its effects are, as well as possibly other information about the world. Using this knowledge, in any situation, the agent can decide what to do by projecting forward various possibilities for actions, and choosing among them one that would lead to achieving its goals. An algorithm that performs this search, and finds a plan of action, is called a planner. Unfortunately, various forms of the planning problem are computationally hard [11, 21, 7]. On the positive side,

*Research supported by ONR grant N00014-95-1-0550, and ARO grant DAAL03-92-G-0115.
sophisticated domain independent search methods that improve performance have been introduced (e.g. [54, 6, 23]) however the range of problems that can be solved is still small.

A natural approach to overcome computational difficulties is to incorporate domain specific knowledge, either coded by hand or automatically extracted, so as to reduce the search time, and several systems on this line have been constructed [28, 33, 15, 53, 2]. Typically these approaches are based around a search engine, and encode control knowledge in some way so as to direct the search mechanism. A variety of methods to extracting control knowledge including EBL [28, 33], static analysis [15], and analogy [51, 52] have been used in this approach, and indeed show some success in several planning domains.

Another response to the difficulty of the problem is to abandon search altogether and construct a special algorithm for a planning domain. This algorithm can be thought of as a mapping from any situation and goal specification to an action to be taken towards achieving the goal, and the approach has thus been called reactive planning. Schoppers [41] suggests to construct such a universal plan by planning for all contingencies and representing the result in some compact way. Another technique to constructing such reactive plans that has proved useful in many problems is that of dynamic programming [3]. This technique can be used in reinforcement learning even when a model of the environment is not available, notably by using temporal difference methods [45, 22].

In this framework learning is typically unsupervised and the agent learns by trial and error in the environment, generalizing from the results of its own actions.

This paper reports on an application of supervised learning to problems of acting in the world, and has been motivated by previous theoretical results [26, 47] regarding such problems. In [26] a notion of pac learning appropriate for (generalized stochastic) planning problems is defined. In this model, a learner can observe a teacher solving problems in a fixed planning domain, and is required to find a \textit{strategy} that can solve problems in the same domain. It is shown [26] that results on Occam algorithms generalize to this model, and that a particular class of strategies, called PRS, akin to decision lists whose rules are existentially quantified first order expressions, can be learned; if a teacher uses a strategy in the class, a learner can find a strategy consistent with the problem solutions that have been observed, and is guaranteed to have good performance. The class of strategies considered is reactive and no search is performed in solving new problems. Instead, the condition-action rules that are learned, explicitly indicate which actions to take in the next step, and are repeatedly applied until the problem is solved (or fail after some pre-specified time bound). Thus, the approach suggests supervised learning of "stand alone" strategies for acting in a dynamic world.

The current paper tests and elaborates on several aspects regarding the applicability of these results for (deterministic) planning problems. First, the assumption on existence of consistent strategies taken in [26] is relaxed. In some of our experiments the examples are drawn by using a planner to solve small problems in the domain. Clearly, when using a planner we have no guarantee that there is a PRS strategy consistent with the actions it chooses. Furthermore, in some domains as is the case for the blocks world we actually have a guarantee that it does not hold since the strategies considered in this paper have no internal state. The class considered in [26], however, allows for a limited use of internal state.

---

1Some arguments were held regarding the utility of this approach; for some views and related technical results see [42, 17, 8, 43].

2A strategy is simply an algorithm for the planning domain. In order not to confuse learning \textit{algorithms} and \textit{algorithms} for a planning domain we refer to the latter as strategies.

3We take reactive to mean having an efficient decision procedure for choosing actions that does not use explicit deliberation on the effects of actions. The term reactive has also been taken to mean having no internal state and thus acting in the same way whenever an identical input is given (notice that this may also be the case for planners). The strategies considered in this paper have no internal state. The class considered in [26], however, allows for a limited use of internal state.
problem is NP-Complete [21]. What is needed is some robustness to this phenomenon.\(^4\) The second issue addressed is that of efficiency. While the bounds in [26] are polynomial they are still rather high. We explore several practical issues that make this application possible, including the use of action models by the learning algorithm, incorporation of "background knowledge" in the form of additional predicates in the domain, and efficient enumeration of candidate rules. We also address the issue of expressiveness. Clearly, for these ideas to be useful, the class of PRS strategies must be expressive enough to encode reasonable algorithms for the problems. We show that for the domains studied here such strategies can be found.

We have experimented with two domains: The blocks world domain has been widely studied before, and due to recent studies [21, 9, 43, 44] its structure is well understood so as to enable thorough analysis. The Logistics transportation domain [51] is more complex and thought of to be closer to real world problems, and has been recently studied from several perspectives [51, 53, 14, 23]. For each domain random problems are drawn and presented with their solutions to the learning algorithm. The learning algorithm uses a variant of Rivest's algorithm [39] to produce a PRS strategy represented as a first order form of decision lists. This strategy is then tested by solving new random problems (of various sizes) in the domain.

In order to produce the training examples we had to provide solutions to the initial set of examples. Solutions were provided either by using a planner (in particular GraphPlan [6]) or by using a hand coded strategy for the domain.

As the experiments demonstrate, generalization is achieved so that unseen problems can be solved by the learned strategies. The strategies produced are not optimal, and in contrast with domain independent search engines they are not complete. That is, they fail to solve some fraction of problems even if given an arbitrary amount of time. However, they are fast, their solutions are not far from optimal, and perhaps most importantly, they can be used to solve some fraction of large problems; problems that are beyond the scope of domain independent techniques. The results are competitive with other techniques and may well be better for large problems. On the other hand, since our strategies may fail, a search engine may be needed in case completeness is required.

The various experiments presented show that the learning algorithm is robust to some extent, that it can be made efficient for small planning domains, and that the class of strategies is indeed expressive enough to handle such domains. We conclude that the approach of learning PRS strategies for planning domains is indeed feasible, and may lead to significant improvement in performance.

We also report on preliminary experiments with strategies based on linear threshold elements. In the propositional domain threshold elements are at least as expressive as simple decision lists, and learning algorithms for them have been developed that are robust against irrelevant attributes and noise [29, 30]. This suggests that it might be useful to have strategies based on weighted sums of condition action rules as used in the PRS. We discuss the implementation of these ideas and preliminary experiments that found these algorithms to perform less well than expected.

As mentioned above this work is related to several previous approaches to learning and planning, and in particular to work in the PRODIGY system, as well as work on EBL, reinforcement learning, and ILP. We discuss these further after presenting the results so as to make the differences clear.

The rest of the paper is organized as follows. Section 2 describes the system, and its interface.

---

\(^4\)Intuitively, one might want to have an agnostic learning result [25], namely an algorithm that finds a strategy that is closest to the examples seen. However, there are no known positive results in this model even for propositional decision lists. In fact, one can conclude from results in [25] that this problem is as hard as learning DNF in the pac model, and that the best that can be achieved is the error rate that can be tolerated for malicious errors. On the positive side one can show that a modification of Rivest's algorithm does achieve a similar bound; namely if the best decision list has error \(\epsilon\) then the algorithm will find a decision list with error at most \(n\epsilon\), where \(n\) is the maximal number of rules on a list.
Section 3 discusses some techniques we use to reduce the complexity of the task. Section 4 describes
the experimental setup and results. Section 5 discusses the use of threshold algorithms. Section 6
discusses related work, and Section 7 concludes.

2 The System

The system, I.2Act [27], includes a learning component and a test component. The learning
component receives as input a description of the domain and traces of solved problems, and outputs
a strategy for the domain represented as a set of rules. The test component receives a strategy and
a list of problems in the domain and applies the strategy to these problems.

2.1 The Input

We developed the system so as to work with the planner GraphPlan written by Blum and Furst
[6], and thus our inputs are based on that system.

The input to the learning algorithm describes a planning domain, problems in the domain, and
their solutions. Examples of the input for the blocks world, and logistics domain can be found in
[27]. The description of the planning domain includes the names of predicates, and models of the
actions given in a standard STRIPS [16] language. Then (an optional part includes) a set of forward
chaining rules that introduces and computes new predicates that we refer to as support predicates.
One can think of these rules as additional background knowledge supplied to the learner. In the
blocks world domain we may have:

Base Rule: on(1,2) on(1,2) on(1,2) => above(1,2)
Recursive Rule: above(2,3) on(1,2) on(1,2) => above(1,3)

The rules come in pairs each introducing a new predicate. The second rule in each pair is
allowed to be recursive and is applied repeatedly until it produces no changes. In the above rules
the numbers 1,2,3 serve as object variables that can be bound to any object in the scene. This
representation is the one used in [26]; in principle, however, any set of monotone forward chaining
rules can be supported.

Then, a set of runs, or complete solved problem, follows. A run is composed of a list of objects,
a list of propositions that hold in the start situation, a list of propositions that should hold in the
goal, and finally a list of actions that achieves the goal.

2.2 Representation of Strategies

The learning algorithm produces as output an ordered list of existentially quantified rules. Rule
based systems have been used before by many and are motivated by work on production systems,
conditioning, neural systems. Following [26] we refer to this class as a PRS, alluding to its relation
to production rule systems. The particular representation we use is exemplified by the rules that
follow. As above, the the numbers 1 and 2 signify object variables. The predicate G is a marker
for goal conditions; that is G(on(1,2)) means that in the goal, the block bound to variable 1 should
be on the block bound to variable 2. The symbol ^ stands for negation.

Rule: OBJECT(1) OBJECT(2) clear(2) holding(1)
G(on(1,2)) G(on-table(2)) on-table(2) => STACK(1,2)

\[^5\text{Here and elsewhere in the representation a predicate may appear several times in a rule. This is simply an issue of convenience of implementation, and is done since we align all rules internally to have the same size, repeating the last predicate whenever needed.}\]

\[^6\text{This non-standard symbol is used since the system manipulates strategies as text files.}\]
The PRS is an ordered set of rules, and its semantics is as follows: As in decision lists [39], the first rule on the list that matches the current example is the one to choose the action. In order to test whether a rule matches the example, we try all possible bindings of objects in the scene to object variables. The first binding that matches (in some lexicographic ordering) is the one to decide on the actual objects with which the action is taken.

The conditions of the rules represent a conjunction of relational expressions. It is sometimes useful to require that two object variables should not bind to the same object. Our system includes an option to enforce this requirement (on all rules) and we used this option in all the experiments reported here. Notice that this implicitly introduces inequalities on all variables and thus increases the expressiveness of the system. As a side effect when using this option some rules on a PRS have to be repeated with the names of variables altered if co-designation is desired.

One might wonder whether this representation is at all appropriate for describing algorithms. While the representation is simple, the ordering of rules with a set of priorities makes for an expressive language. In particular, since the second rule is only applied if the first rule does not match the current situation, the condition of the second rule has in effect a universally quantified (disjunctive) expression conjoined with its own condition (that is the negation of the first rule’s condition). Clearly, rules deeper in the list have an even more complex condition.

As mentioned above, we considered two domains that have been discussed in the literature; the blocks world, and the logistics domain. For these domains, it is in fact not too difficult to write algorithms that solve any problem, and produce reasonable plans, and we present such strategies in Appendix A. This addresses to some extent the issue of expressiveness of the class of strategies, showing applicability to planning domains of interest. In general, it seems to be a question of the predicates used. If the set of predicates is rich enough then it is probably possible to write some strategy in the language. The number predicates that are needed in the condition of the rules and the number of free variables used effect the complexity of learning and execution considerably. It is thus worth noting that the strategies for these domains use small constants for these values.

2.3 The learning Algorithm

We now sketch the learning algorithm that has been used, which is essentially Rivest’s [39] algorithm for learning decision lists. We have also done some preliminary experiments with a generalization of linear threshold algorithms; these are described in Section 5.

The algorithm considers each state that is encountered in any of the runs together with the action taken in this state as an example. It then tries to find a PRS that correctly covers most of these examples. Thus, we take a standard supervised concept learning approach and ignore the history that led to the current state. The same thing happens when using a strategy to choose an action, and therefore our strategies can be classified as being reactive.

A high level description of the algorithm is given in Figure 1. Assume for the moment that the set of all possible rules under consideration can be enumerated. The algorithm first evaluates all the rules on all the examples, marking whether the rule covers the example and in case it does whether it is correct on the example. As discussed above, when more than one binding matches for a rule we choose the first one in some lexicographic order.

The algorithm then chooses among the rules, one that is most preferable according to some preference criterion. This rule is the next rule in the priority list PRS that is produced. Then the
For all rules
   For all examples
      Enumerate all possible bindings
      if the binding matches the condition of the rule then
         • mark that the rule covers the example
         • test and mark whether the rule is correct on the example
         • continue to the next example

While the set of examples is not exhausted
   Choose the most preferable rule
   Add it to the end of the PRS
   Remove all examples that are covered by this rule

Figure 1: The Learning Algorithm

set of examples that are covered by this rule is removed, and the process is repeated until the set of examples is exhausted.

2.3.1 Preference Criteria

As in the case of propositional decision lists [39] if the input is produced by a PRS, we are guaranteed that at least one of the rules is correct on all the examples that it covers. Rivest’s algorithm picks any of these rules.

In case the input is not produced by a PRS, the situation is less clear. In particular, say we have one rule that covers and is correct on exactly one example, and another that covers 89 and is correct on 88; which one should we choose? We have experimented with several preference criteria for choosing between rules, and found that the following (somewhat conservative) criterion does as well or better than others. The criterion simply prefers rules with higher \( \frac{\text{correct}}{\text{cover}} \) ratio, and in case of a tie it prefers the rule that covers more examples. Notice that in the above example it would prefer 1/1 to 88/89. With this choice the algorithm coincides with Rivest’s if there is always a rule consistent with the examples. It is also known to tolerate “classification noise” [24], and as mentioned above one can prove some mild agnostic learning result for it.

2.4 Testing

The system also includes a test component that gets as input a description of the domain, a PRS, and a set of runs. The program tests whether the PRS solves the given problems and in case a problem is solved it also computes the ratio of the solution length to the length of solution given in the run (in case it is given). Thus, we can test what fraction of the problems are solved, and whether the solutions produced are of good quality.

The test component can also be used as a programming environment for PRS. One simply writes down a PRS and tests it to see whether it indeed works well on the domain. This simplifies the otherwise tricky task of actually writing correct PRS strategies.

3 Reducing Complexity

In the discussion of the algorithm above we have ignored the source or structure of the rules under consideration. Recall the rule structure mentioned above:
Rule: \textit{OBJECT}(1) \textit{OBJECT}(2) clear(2) holding(1) \\
\textit{G(on}(1,2)) \textit{G(on-table}(2)) \textit{on-table}(2) \Rightarrow \textit{STACK}(1,2)

One can quantify the number of rules under consideration with two parameters: $k_R$ the number of elements in the left hand side of the rule, and $k_B$ the number of free variables allowed in the rule. Let the arity of predicates be bounded by $k_A$, and the number of predicates be $n$, then for each action the number of rules that corresponds to $k_R$, $k_B$ is at most $(4n)^{k_R} k_B^{k_B} k_A^A$ (since each predicate can appear either positive or negated and either with a goal marker $G$ or not, and we need to choose the names for the variables). Thus for small parameters we may be able to enumerate all this class of rules. However, this number grows exponentially fast in $k_R$.

Another source of complexity is the problem of binding. If a rule has $k_B$ variables, then when confronted with a situation with $k_O$ objects, in principle one has to test all possible bindings of the variables, that is $k_O^{k_B}$ possibilities, namely exponential in the number of free variables.

It is instructive to compare the relational learning problem to the propositional counterpart, were we to fix the size of the domain. In the propositional case object names must be explicitly stated in the rules, and the number of rules for each action becomes $(4n)^{k_B} k_O^{k_B} k_A^A$. By fixing $k_B$ to a small constant we in fact reduce the number of rules and the size of hypothesis class. Thus, the utility of the relational learning formulation is not only in the convenience of representation, or in the fact that the results of learning apply to problems of different size, but also in reducing the size of the hypothesis space and the complexity of the learning problem.

While in the worst case one may have to endure these complexities, there are various possibilities for reducing them considerably in practice. In the first place, one can make the enumeration more efficient, by not duplicating rules, and by not using rules that are useless (for example if they always lead to a contradiction). For the problem of enumerating bindings one can remember why a previous binding did not match and not produce another binding that fails for the same reason. Several other simple techniques are used in the system; we discuss two that are particularly relevant for the problem of planning.

Recall that as part of the input we get models of the actions in STRIPS form, each containing a set of preconditions, such that the action can be taken only if the preconditions hold. Therefore, any rule that recommends a certain action should have its preconditions on the left hand side. In the example above the first line includes the preconditions of the action \textit{STACK}; this effectively reduces the part of the rule that is enumerated from 7 to 3, and reduces the complexity considerably. Thus, action models help in focusing the learner in its search for good rules. Clearly, an implementation of this is straightforward.

The other technique we use is to look for information in the runs themselves and enumerate only rules that have a chance to be useful. For example, we search the runs and mark all the predicates that ever appeared as part of the goal. Clearly, it is not necessary to have \textit{G}(p(x)) in the rule if $p()$ never was a part of a goal. A more dramatic application of this technique can be seen in domains where types play an important role. For example in the Logistics domain there are objects of several types including \textit{OBJECT}, \textit{TRUCK}, \textit{AIRPLANE}, \textit{CITY}, as well as the predicate \textit{in}(x,y). By searching the runs we observe that the predicate \textit{in}() accepts only \textit{OBJECT} in the first parameter and only \textit{TRUCK} or \textit{AIRPLANE} in the second parameter. Now clearly one should not try to use a rule with a construct like \textit{AIRPLANE}(x) \textit{CITY}(y) \textit{in}(y,x). In general every variable must have at least one type common to all its occurrences. This is easily achieved and reduces the number of rules considerably. Furthermore, with a bit of book-keeping this also helps in the problem of binding enumeration, since unreasonable bindings do not need to be considered in the first place. We note that in the blocks world there is only one type of objects and thus the technique is not applicable.
Another application of this technique follows Valiant’s [48] idea suggested in the context of learning DNF expressions. There, one first enumerates conjunctions that appear in the examples (with some frequency), and only then tries to learn a disjunction of them. This problem has been recently studied in data mining in the context of mining association rules [1]. A simple bottom up enumeration algorithm enumerating the so-called frequent sets has been shown useful in practice as well as to be optimal in some special cases [1, 32, 20]. Following [32], we refer to the algorithm as the levelwise algorithm. The algorithm starts by enumerating elements of size one. Then, of the elements of size \(i\) that were found frequent, it constructs candidates of size \(i + 1\) and tests whether they are frequent. A version of this algorithm adapted to the problem of learning PRS is used in the system. This technique adds one more parameter to the setup of the system, namely the frequency threshold used. Naturally, the threshold should depend on the average solution length (since a rule that is used once in every solution should be included). This is discussed further with the experimental results.

These techniques reduce the number of rules considered by several orders of magnitude as well as reducing binding time. The complexity is however still somewhat high. We have so far tested the system for the blocks world with \(k_R = 2\) to 3, and \(k_B = 3\). For the logistics domain, the additional type checking allows testing with \(K_r = 3\) and \(k_B = 5\). For reference, our hand written PRS for the blocks world had some rules with \(k_R = 5\) and \(k_B = 4\), and the one for the Logistics domain had \(k_R = 4\) and \(k_B = 6\).

4 Experimental Results

4.1 The Blocks World Domain

In this domain, a set of cubic blocks is arranged on a table, and one has to move them from one configuration to another. The set of operations include: PICKUP(\(x\)), PUTDOWN(\(x\)), UNSTACK(\(x, y\)), and STACK(\(x, y\)) with the obvious semantics. The predicates are on(\(x, y\)), clear(\(x\)), on-table(\(x\)), holding(\(x\)), and OBJECT(\(x\)). A planning problem in this domain includes an arrangement of blocks in the current situation, and a list of required goal conditions (say, it is required that block 1 is on 2), that does not necessarily describe a complete situation\(^7\) (the position of block 2 may be unspecified). It is known that the problem of finding the smallest number of operations needed in this domain is NP-complete, and that there are simple algorithms that use at most twice the minimum number of steps [21]. Thus, it is easy to write an algorithm for the domain that while not optimal performs quite well. The challenge of planners is to use a general technique and solve the problem using it. Currently, the best domain independent planners can solve problems with 10 to 12 blocks. (See [44, 2, 23] for recent relevant work with various alternative techniques).

4.1.1 Experimental Setup

Our learning algorithm receives as examples problems and their optimal solutions. We generate examples in the following manner (1) Random blocks world states are generated using \texttt{bwstates} a program written by Slaney and Thiebaux [44]. (2) Pairs of states are translated into planning problems. We chose to have the goal partially described. For this purpose the location of a third of the blocks is omitted in the goal. (3) The planner GraphPlan written by Blum and Furst [6] is used to solve the problems. We also supply the algorithm with knowledge about the domain in the form

\(^7\)We note that the fact that the goal is only partially specified makes the problem a bit more tricky, since a partial stack of blocks all of whose goal conditions are satisfied may need to be moved if it is above another block that is not in its goal state.
of support predicates, as explained in the previous section. In particular the following predicates were given:

**Base Rule:** \( G(on(1,2)) \) \( G(on(1,2)) \) \( G(on(1,2)) \) \( \implies \) \( \text{ingoal}(1) \)

**Recursive Rule:** \( G(on-table(1)) \) \( G(on-table(1)) \) \( G(on-table(1)) \) \( \implies \) \( \text{ingoal}(1) \)

**Base Rule:** \( G(on-table(1)) \) \( on-table(1) \) \( on-table(1) \) \( \implies \) \( \text{inplacea}(1) \)

**Recursive Rule:** \( \text{inplacea}(2) \) \( G(on(1,2)) \) \( on(1,2) \) \( \implies \) \( \text{inplacea}(1) \)

**Base Rule:** \( G(on(1,2)) \) \( on(1,2) \) \( \neg \text{ingoal}(2) \) \( \implies \) \( \text{inplaceb}(1) \)

**Recursive Rule:** \( \text{inplaceb}(2) \) \( G(on(1,2)) \) \( on(1,2) \) \( \implies \) \( \text{inplaceb}(1) \)

**Base Rule:** \( on(1,2) \) \( on(1,2) \) \( on(1,2) \) \( \implies \) \( \text{above}(1,2) \)

**Recursive Rule:** \( \text{above}(2,3) \) \( on(1,2) \) \( on(1,2) \) \( \implies \) \( \text{above}(1,3) \)

Notice the predicates **inplacea** and **inplaceb** give information that is only partially useful for the task. (**inplacea** is only useful for goal stacks that start on the table, and a block that is **inplaceb** may still need to be moved.)

Using the above method we generated example problems, all of which included 8 blocks, and trained the algorithm with \( k_R = 2 \) and \( k_B = 3 \). Similarly, random test problems with varying number of blocks were generated. Unless otherwise specified all experiments used the levelwise algorithm with \( \sigma = 0.01 \) as threshold.

The learning time for these experiments was roughly 130 minutes (for training with 4800 examples each with 8 blocks), on a SUN/20 workstation. The learning time grows roughly linearly with the number of examples (since the dominating factor is the time to evaluate the rules). We also ran some experiments with \( k_R = 3 \) and \( k_B = 3 \); naturally learning was slower but still feasible. This however did not improve the results and we thus omit the details.

### 4.1.2 Results

Figure 2 describes the fraction of problems solved by the output of the learning algorithm as a function of training size, and for several problem sizes. For each sample size, each data point in the graph represents an average of five independent learning experiments, and the ratio of solved instances has been estimated using 400 independent runs (for block sizes 7, 8, 10) or 100 independent runs (for block sizes 12, 15, 20). In the graph and similarly in other figures to follow the training size refers to the number of situation action pairs, rather than to complete problems. The number of complete problems corresponding to the 4800 situation action pairs was roughly 315.

As one can observe from Figure 2 generalization is achieved, and a significant fraction of the problems is solved by the strategies. Roughly 80 percent of problems of the same size (8 blocks), and 60 percent of large problems (with 20 blocks) are solved. For comparison, on a random sample of 10 instances with 12 blocks each GraphPlan solved 2 in less than half an hour. We can also see that the performance stops improving well before using all the sample. Namely, the sample size is sufficiently large. As observed in other studies the sample size needed is smaller than the corresponding pac-learning bounds. The figure also includes plots of the variation observed in the success of strategies learned,\(^8\) which is somewhat high. Of the 5 experiments averaged the best run produced strategies that solved 86 percent of problems with 8 blocks, and 76 percent of problems with 20 blocks.

---

\(^8\)This was computed as the “measured standard deviation”, \( \sqrt{\frac{1}{n} \sum (s_i - \bar{s})^2} \), where \( s_i \) denotes the success in the \( i \)th experiment, and \( \bar{s} \) is the average success. For reference, the standard deviation for the estimates for each separate strategy, namely for 100, 400 Bernoulli trials is bounded by 0.05, 0.025 respectively.
It is worth noting that the PRS strategies are efficient; the time for solving a problem with 8 blocks is less than a second, and for 20 blocks it is well less than a minute. The strategies do get slower for larger problem sizes, however time required grows polynomially, where the dominant factor is the matching time $k_{RB}$.

Figure 3 shows how the success rate scales with the size of the problems (for several training sizes). We can see that performance goes down with the number of objects in the problem but that it degrades gracefully still solving problems of larger size. To test this issue further we applied the strategies learned on 4800 examples to a set of 100 problems each with 50 blocks. The average success rate (of the 5 experiments) was 16 percent. The variance was again high the best strategy solving 38 percent and the worst solving 4 percent.

Some fraction of problems remains unsolved by the learned PRS. In this respect note that we took a simple approach in evaluating the learned strategies. In particular, in many cases, on problems on which a PRS fails, it arrives in a state of self-loop where the same action is done and undone repeatedly. This is a situation which is easily identified, and one could in principle escape from this situation, and improve the performance, by choosing a random action and then restarting the PRS. In order to have a fair evaluation of the deterministic PRS we have not done so. Notice that while there are PRS strategies that can solve all problems there is no such strategy that is consistent with the examples given to the learner. Thus the learning algorithm in some sense finds heuristic rules that come close to the actions taken by the planner, and therefore fails in some cases.

4.1.3 Quality of Solutions

So far we only discussed the fraction of problems solved but ignored the quality of solutions; here quality can be measured as the number of steps in the solution. For problems of small size (7 and 8 blocks), where we could use the planner to solve a large number of problems, the solutions produced by the PRS were consistently close to those of the planner (less than 10% increase in length).
The blocks world domain has been extensively studied, and an experimental evaluation of several approximation algorithms has been recently performed. In particular Slaney and Thiebaux [14] identify three versions of the approximation algorithm that guarantees at most twice the number of optimal steps. The first algorithm called US first moves all misplaced blocks to the table and then constructs the required towers. The second algorithm called GN1, improves on that by checking whether it can move a block to its final position in which case it does so, and otherwise it moves an arbitrary misplaced block to the table. Thus GN1 reduces the number of the steps by avoiding some of the intermediate moves to the table. A third algorithm GN2 improves further by cleverly choosing which misplaced block to move to the table. In their study Slaney and Thiebaux compare the solution lengths produced by these algorithms against each other and against the optimal solutions. The algorithms US and GN1 can be easily coded as PRS strategies and we can thus compare their performance to that of the learned strategies.

Figure 4 plots the ratio of solution lengths produced by the learned strategies to those of US and GN1 for two problem sizes (on a single learning experiment). As can be seen the learned strategies perform better than these algorithms. The ratio against GN1 is 0.98 for 8 blocks and 0.96 for 20 blocks. The ratio against US is 0.91 for 8 blocks and 0.87 for 20 blocks. Figure 5 concentrates on GN1 plotting the average performance ratio over the 5 experiments, and shows that indeed the learned strategies perform better than GN1 for all block sizes, and that for larger problems the difference between the learned strategies and GN1 becomes more pronounced, from 0.99 on 7 blocks to 0.95 on 20 blocks. Since GN1 was found to be close to optimal in practice in [44] we can conclude that the learned strategies are indeed of high quality.

4.1.4 Preference Criteria

As mentioned above we have experimented with several preference criteria for choosing between rules, and the one used in the above results is the conservative one preferring consistency over high
coverage. One other criterion that seems to provide comparable performance is the following: Fix some accuracy degree $\gamma$ (say $\gamma = 0.9$). A rule with accuracy greater than $\gamma$ is preferred to one with accuracy smaller than $\gamma$. If two rules both have accuracy greater than $\gamma$, then prefer the one that covers more examples. Notice that in the example given in the previous section this criterion prefers the rule with 88/89 to the one with 1/1. To illustrate this issue Figure 6 compares the performance of the standard criterion (denoted PF0) with this new criterion (denoted PF2) on a single learning experiment and two problem sizes. Indeed we see that the performance is rather close and that neither dominates the other. Naturally, differences in performance may arise in other domains, and the second performance criterion may prove useful.

4.1.5 Threshold for Enumeration

All the experiments reported above used the bottom up levelwise algorithm in enumerating the candidate rules, with $\sigma = 0.01$ as threshold. Namely a rule was deemed worthwhile if it covered at least one percent of the original sample. Clearly the threshold can effect the performance drastically. On one hand, if the threshold is very low then we should expect the number of rules to be high, and the performance to be close to the one with a standard enumeration of rules. On the other hand, if the threshold is too high then important rules will be missed and performance will decrease considerably. Figures 7 and 8 plot the performance of learned strategies on single learning experiments each with a different enumeration (including the simple enumeration not using the levelwise algorithm) but with the same examples, on test problems with 8 and 20 blocks respectively. Indeed in both cases a threshold of $\sigma = 0.001$ (which means the a rule had to be correct on 5 of the 4800 examples) performs similar to the standard enumeration, and $\sigma = 0.05$ leads to bad performance. For comparison the average solution length in these runs is 15.2 steps (so that a rule that is used once every two runs might be discarded with $\sigma = 0.05$). It is interesting to notice that the intermediate values lead to somewhat more smooth graphs perhaps by filtering rules.
that perfectly cover a very small number of examples and are therefore chosen by the conservative preference criterion.

The reduction in the number of rules and hence running time is less regular than one might expect. Figure 9 plots the performance, as well as the number of rules enumerated as a function of \( \sigma \) (the number is normalized where 1 corresponds to 3390). We can see that the number of rules falls immediately with \( \sigma = 0.001 \) and decreases only slightly with larger values. On the other hand the performance falls drastically only with \( \sigma = 0.05 \). Obviously the important rules lie somewhere in the intermediate values, but this raises the question of why the number of rules falls so slowly after the initial step. A reasonable explanation is the existence of rules with “spurious” conditions that hold in every situation. These rules will not be filtered by any threshold but on the other hand they are not useful for the strategies. We expect that clever filtering of such spurious rules can lead to improvements in the enumeration and performance of the learning algorithm.

4.2 The Logistics Domain

The Logistics transportation domain introduced by Veloso [51] is a somewhat more complex domain where one has to ship packages between several locations through the use of trucks and airplanes (for locations in the same city one has to drive and for inter-city transfers one has to fly). There is a limited number of trucks and airplanes and the task is to minimize the number of operations needed to ship the packages. The domain is an abstraction of a real transportation domain and has been recently studied in several frameworks [23, 14, 52, 53].

The domain includes the unary predicates OBJECT, TRUCK, LOCATION, AIRPLANE, AIRPORT, CITY that indicate some information about the “type” of objects (types however are not unique and some objects for example may belong to more than one type e.g. to both LOCATION and AIRPORT). The domain further includes the predicate \( \text{at}(x,y) \) indicating the location of objects and vehicles,
\textit{in}(x, y)$ for indicating that some objects are in some vehicle, and \textit{loc-at}(x, y)$ to indicate in which city a location resides. The domain also includes the actions \textsc{load-truck}, \textsc{load-airplane}, \textsc{unload-truck}, \textsc{unload-airplane}, \textsc{drive-truck}, \textsc{fly-airplane} with the expected definitions. A complete description can be found in [27].

4.2.1 Experimental Setup

For this domain we drew random problems from a fixed subset as follows. In all problems we fixed the number of cities to 3, the number of trucks to 3 (one in each city), the number of locations in each city to 2 (one of them being an airport), and the number of airplanes to 2. The location of airplanes, and the location of a truck within the city were randomly chosen. The number of packages was varied and their locations in the starting position and goal position were randomly chosen. All packages were assigned a goal position.

For the training we generated examples with 2 packages, and for testing we generated several sets of problems with 2, 6, 10, 15, 20, and 30 packages respectively. The number of objects in each training example was thus 16, and generally it was 14 plus the number of packages.

4.2.2 Results

Preliminary experiments using GraphPlan [6] revealed that the output of the planner is too varied and does not fit any PRS with the strict ordering of bindings and rules. (The prediction on the training sets was about 60 percent correct.) We therefore generated examples using a PRS that was hand coded. This PRS has $k_R = 4$ and $k_B = 6$ and it produces solutions of comparable length to those of GraphPlan. The learning algorithm was run with $k_R = 3$ and $k_B = 5$; thus here again there is no strategy in the class that is consistent with the examples and the robustness of the algorithm is tested.
We note that in this domain some of the actions have arity 4 and the number of rules is initially very large. Nevertheless we were able to run experiments with parameters larger than the blocks world domain due to the pruning method that uses type checking, and reduces the number of rules by several orders of magnitude, as well as reducing the required match time. Here again we see that adding structure to the problem makes the learning problem more manageable by reducing the number of possible rules and thus the size of the hypothesis class.

Figure 10 plots the success rate of the learned strategies (averaged over five experiments) on the various test problem sizes. As in the blocks world domain we see that learning is successful and that a considerable fraction of test problems is solved. Here again for comparison we ran GraphPlan on 10 of the test problems with 15 packages, and only 1 was solved in less than half an hour. In contrast for the problems solved of this size the PRS took less than 20 seconds.

An interesting phenomenon occurs where the success rate is not monotonically decreasing with the number of packages in the problem. This may be due to the fact that when there are few packages we are likely to have a location for which nothing is sent but from which one has to fetch packages. On the other hand with many packages it is likely that one has to visit all locations anyway. Thus tricky situations happen more rarely and the problem becomes easier for reactive strategies.

We have also performed some experiments in which the training set included problems with 6 packages; these yielded similar results though with lower rates of success (around 40 percent) when trained with the same number of examples. This may be explained by the fact that with more packages the run length is larger; therefore the same number of situation-action pairs correspond to less runs in the larger domain and thus provide less information.

While we have not done an extensive comparison, the length of solutions was found to be comparable to the solutions of GraphPlan on problems with 6 packages.\(^9\) We conclude that successful

---

\(^9\)GraphPlan finds a “shortest parallel solution” but not necessarily a shortest solution. Tested on the 400 test
Another thing to observe in Figure 10 is the rather high variance in success rate. This essentially results from the fact that one of the five experiments produced strategies with a rather low performance; otherwise the other four experiments produced quite similar results. This problem of high variance occurred to some extent also in the blocks world and its prevention is an important issue to be addressed.

5 Multiplicative Update Threshold Algorithms

Linear threshold elements are at least as expressive as decision lists, and they possess some useful learning algorithms. In particular, the Winnow, and the weighted majority algorithms [29, 31] have been shown to have good theoretical properties, and recently this has been validated in experiments in various domains [30, 5, 18]. Valiant [49, 50] suggests that action strategies can be embedded in the Neuroidal architecture thus taking advantage of the robustness of threshold elements and their algorithms. In this section we discuss preliminary experiments adapting these ideas to the problem of learning to act.

The Winnow algorithm [30] is designed for binary classification problems. It is similar to the perceptron algorithm, the only difference being that the update to the weights is multiplicative rather than additive. The algorithm maintains a weight for every attribute and predicts 1 if the weighted sum of the attributes that have value 1 is above a threshold. In case a mistake has been made it updates the weights of attributes that have value 1, multiplying or dividing by a constant.

In the weighted majority algorithm [31] the attributes are thought of as “experts”, and the algorithm predicts 1 on a given example if the weighted sum of the attributes that have value 1 is

---

runs with 6 packages the average ratio of solution lengths was found to be 0.99, our PRS solutions being shorter on average.
larger than the weighted sum of the attributes that have value 0. In case a mistake has been made, the algorithm updates the weights, multiplying by some factor.

One difference between the two algorithms is in the update. Winnow changes only the weights of the attributes that hold in the current example (increasing in case of false negative and decreasing otherwise). Weighted majority decreases the weights of the side that voted incorrectly. Another difference concerns the voting; while in the weighted majority all attributes contribute to some vote, in Winnow attributes that have value 0 do not participate in the vote.

We have experimented with several variants of these algorithms and we briefly discuss the one that proved most useful (which is somewhat similar to the variant used by Blum [5]). For our problem there are many possible outputs, and their number is not fixed in advance (and changes for example with the number of objects in the problem). Thus, a threshold mechanism for 2 classes is not sufficient; instead we use a weighted voting scheme similar to the weighted majority algorithm. Our algorithm considers each rule enumerated as a basic attribute, and maintains a weight for each such rule. Given a new situation, each rule is evaluated and in case it matches the example the action produced by that rule (its vote) is added to a list of votes with the rule’s weight. The algorithm then chooses as output the action that received the maximum weighted vote. When a mistake is made we update the weights of rules that covered the example, similar to what is done in Winnow. For the purpose of update we use a fixed threshold. If the sum of weights of rules that voted the right action is less than the threshold then they are increased. If the sum of weights of all rules that voted wrong actions is more than the threshold then they are decreased. Notice that the threshold is not used during action selection, but only for the update. We found that without the use of a threshold, the weights may grow in an uncontrolled manner, and produce worse results.

We have tested this algorithm in two forms. In the first we took the same voting scheme as above and applied it to new problems. In the second we sorted the weights and created a PRS based on these weights as priorities. Namely, the rule with the highest weight is chosen to be the first in
the PRS, etc. Interestingly, the second form performed better than the first in all our experiments. It also yields more efficient strategies since the number of rules that have to be evaluated on each example is much smaller. However, both forms had success rates that were much lower than that of our standard PRS algorithm, and we omit the details.

Recall that while linear threshold elements are more expressive than 1-decision lists, the fast convergence of Winnow is not guaranteed for decision lists [30]. In fact simple 1-decision lists may force the algorithm to make an exponential number of mistakes. Another source of difficulty arises since we are using a single threshold for a multi-class problem whose size is not pre-fixed. These issues may explain the lower convergence observed. There are also other possibilities for adapting Winnow to the current relational learning problem by casting it into a two class classification problem, that may prove more successful. However, more experimentation will be needed to clarify these issues.

6 Related Work

This is certainly not the first work to apply ideas of learning to the problem of acting in the world. Several systems have been constructed around the idea of learning control rules. These systems are based around some search (or problem solving) method and various techniques are used to acquire control knowledge that can direct the search. The techniques include EBL methods [12, 34] used in prodigy [33] and in Soar [28, 40], static analysis in Prodigy [15], analogy in Prodigy [51, 52, 53], and ILP in Scope [14]. Our approach clearly differs in the method of acquiring rules, but perhaps more importantly, the strategy that our learning algorithm finds, while in the form of a collection of rules, is not used as a part of a search algorithm, but instead used as a stand alone algorithm for the domain.

While several of these systems have studied the same domains it is not possible to make direct
comparisons for several reasons. First, the input to the various systems is not identical; some of the above mentioned systems use domain axioms while our system uses support predicates. Secondly, since these systems are based around a search engine, performance was measured by the amount of speedup on the same set of problems for which training took place [33, 15, 51]. This set typically includes problems of various sizes, some of them of relatively small size. For example, for the blocks world, Minton [33] (and following him other works in Prodigy) reports on one set of 100 problems with 3 to 12 blocks. Estlin and Mooney [14] report on problems with 2 to 6 blocks. Since these systems were developed several years ago and on different equipment the differences are hard to evaluate. In the logistics domain Estlin and Mooney [14] report on problems with 2 packages. Veloso et. al. [53] report on a system meant to improve the quality of solutions where some problems with 20, and 50 packages are tested. The percentages of success rate would indicate similar performance to the results presented in Figure 10, but one should bear in mind the the evaluation procedures were different. Resorting to qualitative comparison our work indicates that the reactive approach to planning and in particular learning reactive strategies is competitive with other approaches thus substantiating the claim for the feasibility of the approach.

In this area our model is closest to several works by Tadepalli et.al. [46, 47, 38] that combine ideas from speedup learning and supervised learning. Our work extends these efforts in two directions, one being the use of a new representation for strategies (the PRS), and the second being the relaxation on the assumption on existence of consistent strategies.

Another direction that has been pursued is that of reinforcement learning (see [22] for a survey), where a generalized form of stochastic planning problems is studied. In this framework, the agent can act in the world and receives information by being reinforced in some situations. Thus, learning is done in some sense by trial and error, and is unsupervised. Our work is similar to this approach in that strategies produced by learners are do not involve search. The main difference to our work is in that we use examples, and thus study supervised learning, a problem that is in principle easier (since one can always ignore the teacher).

Our work is also similar in nature to work done in ILP [37, 36, 35] where logic programs that consist of first order rules are induced form examples. However, the models differ in details that are crucial. An example in ILP includes a single ground instance of a relation; the rest of the information on this example is provided through the background knowledge. In contrast an example in our system describes a complete situation and the ground action taken in that situation (that can be seen as similar to the relation being learned). Thus, in some sense our examples have “changing background knowledge” in ILP terms. Our support predicates behave similarly to the background knowledge, however they again contribute to the varying state information. Another difference results from the nature of the task. In the planning domain one has to choose a single action on each input. We have thus employed a priority ordering also between bindings. In contrast, in ILP the task is in some sense static, and thus all bindings of a rule (or at least most of them) must produce correct classification. Nevertheless, the structure of induced expressions is similar, and similar techniques can be used in both models, so that some insights from ILP methods may be useful in our system.

---

10 Success rate for this particular instance is reported there under a time bound of 450 seconds in their system.
11 Perhaps the best studied RL algorithm so far is Q-Learning and its variants. This algorithm computes or approximates an estimate of the expected future payoff of taking an action in each situation, and thus uses an architecture different from ours. Our approach is more similar to systems where a genetic algorithm is combined with a reinforcement learning strategy, and the architecture is composed of a collection of rules [4], or competing collections of rules [19].
7 Conclusion

The paper describes experimental results with a system, L2Act, that performs supervised learning of PRS strategies for planning domains. The system incorporates several techniques that allow learning in otherwise too large domains.

Our results for the blocks world and the logistics domains are encouraging, and indicate that the approach is at least in principle feasible, and may lead to significant improvement in performance. Indeed large problems in the domain can be solved after training with small problems, and the solutions found are of high quality. The experiments also exhibited the robustness of the learning algorithm to “noise” in the data, having dealt with situations where no PRS strategy is consistent with the data. However, the applicability is limited to cases where a PRS can explain most of the observed examples; when a planner was used to generate examples for the logistics domain, learning was less successful. We have also partly validated a claim for expressiveness by coding PRS strategies for both domains. In general this seems to be a question of the richness of the predicates used.

It was indicated that by turning to a relational learning problem we have in fact reduced the complexity of the problem; effectively reducing the size of the hypothesis space (relative to the propositional domain) while maintaining an efficient learning algorithm. This is of course not a general feature of relational representations (c.f. hardness of reasoning problems) or of learning in general but a feature of the representation chosen here. The representation is also useful in that several other techniques have been developed for it, thus one may be able to combine such techniques in a single system, taking advantage of opportunities as they arise.

There are several directions for possible future work. First, work on reducing the complexity of the algorithm further is possible. In particular, we mentioned the possibility of pruning useless rules that are nevertheless frequent since their conditions are tautologous. Another source of complexity is the matching process that dominates the learning time. This issue has been addressed before in production systems (see e.g. [13] for recent work) and its improvement can effect the learning time considerably. In this paper we concentrated on the application of a method that is provably correct under some assumptions. Of course other learning techniques might prove useful. In particular the techniques applied by CN2 [10] and FOIL [37] use a similar representation and can be applied.

We have also discussed preliminary experiments with linear threshold algorithms, and this direction suggests interesting questions. The success of our current system, like that of other systems applied to planning problems, relies on the fact that the number of predicates used was small. Any system dealing with a variety of problems will have a large number of predicates many of which may be irrelevant to many of the tasks. The Winnow algorithm [29] is particularly useful for handling irrelevant attributes. Thus a successful application of this approach may be crucial in scaling the system.

Our experiments suggest that supervised learning algorithms can be used for problems of acting in dynamic environments. This offers a new challenge for supervised learning methods in domains where it is relatively easy to get large numbers of examples for training and testing.

Acknowledgments

I am grateful to Les Valiant for many helpful discussions and suggestions, and to Prasad Tadepalli and Bart Selman for comments on earlier drafts. I am also grateful to John Slaney and Avrim Blum for having made their programs, bwstates and GraphPlan respectively, publicly available.
A Hand Coded Strategies

A.1 Blocks World

Rule: OBJECT(1) OBJECT(2) clear(2) holding(1)
inplacea(2) G(on(1,2)) G(on(1,2)) G(on(1,2)) G(on(1,2)) => STACK(1,2)

Rule: OBJECT(1) OBJECT(2) clear(2) holding(1)
\_ingoal(2) G(on(1,2)) on-table(2) on-table(2) on-table(2) => STACK(1,2)

Rule: OBJECT(1) holding(1)
\_ingoal(1) \_arm-empty \_arm-empty \_arm-empty \_arm-empty => PUT-DOWN(1)

Rule: OBJECT(1) holding(1)
G(on(1,2)) above(2,3) inplacea(3) inplaceb(3) ingoal(3) => PUT-DOWN(1)

Rule: OBJECT(1) holding(1)
G(on(1,2)) above(2,3) G(on(4,3)) \_on(4,3) \_arm-empty => PUT-DOWN(1)

Rule: OBJECT(1) OBJECT(2) clear(2) holding(1)
inplaceb(2) G(on(1,2)) G(on(1,2)) G(on(1,2)) => STACK(1,2)

Rule: OBJECT(1) OBJECT(2) clear(2) holding(1)
\_ingoal(2) G(on(1,2)) \_arm-empty \_arm-empty \_arm-empty \_arm-empty => STACK(1,2)

Rule: OBJECT(1) holding(1)
\_arm-empty \_arm-empty \_arm-empty \_arm-empty \_arm-empty \_arm-empty => PUT-DOWN(1)

Rule: OBJECT(1) clear(1) on-table(1) arm-empty
inplacea(2) G(on(1,2)) clear(2) clear(2) clear(2) => PICK-UP(1)

Rule: OBJECT(1) clear(1) on-table(1) arm-empty
\_ingoal(2) G(on(1,2)) on-table(2) clear(2) clear(2) => PICK-UP(1)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
above(1,3) G(on(3,4)) \_on(3,4) arm-empty arm-empty arm-empty => UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
above(1,3) G(on-table(3)) \_on-table(3) arm-empty arm-empty arm-empty => UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
G(on(2,3)) \_on(2,3) arm-empty arm-empty arm-empty arm-empty => UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
G(on-table(2)) \_on-table(2) arm-empty arm-empty arm-empty arm-empty => UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
above(1,4) G(on(3,4)) ^_on(3,4) arm-empty arm-empty ==> UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
G(on(3,2)) ^_on(3,2) arm-empty arm-empty arm-empty ==> UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
^_inplacea(1) ^_inplaceb(1) ingoal(1) arm-empty arm-empty arm-empty ==> UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
^_inplacea(3) ^_inplaceb(3) ingoal(3) above(1,3) arm-empty arm-empty arm-empty ==> UNSTACK(1,2)

Rule: OBJECT(1) OBJECT(2) on(1,2) clear(1) arm-empty
^_inplacea(2) ^_inplaceb(2) ingoal(2) arm-empty arm-empty arm-empty arm-empty ==> UNSTACK(1,2)

Rule: OBJECT(1) clear(1) on-table(1) arm-empty
inplaceb(2) G(on(1,2)) clear(2) clear(2) clear(2) ==> PICK-UP(1)

Rule: OBJECT(1) clear(1) on-table(1) arm-empty
^_ingoal(2) G(on(1,2)) clear(2) arm-empty arm-empty arm-empty arm-empty ==> PICK-UP(1)

A.2 The Logistics Domain

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) in(1,2)
G(at(1,3)) G(at(1,3)) G(at(1,3)) G(at(1,3)) ==> UNLOAD-TRUCK(1,2,3)

Rule: OBJECT(1) AIRPLANE(2) LOCATION(3) in(1,2) at(2,3)
loc-at(4,5) loc-at(3,5) G(at(1,4)) G(at(1,4)) ==> UNLOAD-AIRPLANE(1,2,3)

Rule: OBJECT(1) AIRPLANE(2) LOCATION(3) in(1,2) at(2,3)
G(at(1,3)) G(at(1,3)) G(at(1,3)) G(at(1,3)) ==> UNLOAD-AIRPLANE(1,2,3)

Rule: OBJECT(1) AIRPLANE(2) LOCATION(3) at(1,3) at(2,3)
G(at(1,4)) loc-at(4,5) ^_loc-at(3,5) ^_loc-at(3,5) ==> LOAD-AIRPLANE(1,2,3)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) at(1,3)
G(at(1,4)) loc-at(4,5) loc-at(3,5) loc-at(3,5) ==> LOAD-TRUCK(1,2,3)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) at(1,3)
G(at(1,4)) loc-at(3,5) loc-at(6,5) AIRPORT(6) ==> LOAD-TRUCK(1,2,3)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) at(1,3)
G(at(1,4)) loc-at(4,5) ^_loc-at(3,5) ^_AIRPORT(3) ==> LOAD-TRUCK(1,2,3)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) in(1,2)
G(at(1,4)) loc-at(4,5) ^_loc-at(3,5) AIRPORT(3) ==> UNLOAD-TRUCK(1,2,3)
Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4) in(5,1) G(at(5,3)) G(at(5,3)) G(at(5,3)) \implies\text{DRIVE-TRUCK}(1,2,3,4)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4) in(5,1) \neg G(at(5,2)) \neg G(at(5,2)) \neg G(at(5,2)) \implies\text{DRIVE-TRUCK}(1,2,3,4)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4) at(5,3) G(at(5,6)) \neg \text{loc-at}(6,4) \text{AIRPORT}(2) \implies\text{DRIVE-TRUCK}(1,2,3,4)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2) in(4,1) G(at(4,5)) \text{loc-at}(5,6) \text{loc-at}(3,6) \implies\text{FLY-AIRPLANE}(1,2,3)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2) in(4,1) G(at(4,3)) G(at(4,3)) G(at(4,3)) \implies\text{FLY-AIRPLANE}(1,2,3)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4) at(5,3) G(at(5,2)) G(at(5,2)) G(at(5,2)) \implies\text{DRIVE-TRUCK}(1,2,3,4)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2) at(4,3) G(at(4,5)) \text{loc-at}(3,6) \neg \text{loc-at}(5,6) \implies\text{FLY-AIRPLANE}(1,2,3)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2) at(4,3) G(at(4,2)) G(at(4,2)) G(at(4,2)) \implies\text{FLY-AIRPLANE}(1,2,3)

References


