Learning Action Strategies for Planning Domains

Roni Khardon*
Division of Informatics
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ
Scotland
roni@dcs.ed.ac.uk

September 28, 1999

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 transportation domain, using strategies in the form of a generalisation of decision lists. The condition of a rule in the decision list is an existentially quantified first order expression, and each such rule indicates which action to take when the condition is satisfied. The learning algorithm is a variant of Rivest's [33] algorithm, improved with several techniques that reduce its time complexity. The experiments demonstrate that the approach is feasible, and generalisation is achieved so that unseen problems can be solved by the learned strategies. Moreover, the learned strategies are efficient, the solutions found by them are competitive with those of known heuristics for the domains, and transfer from small planning problems in the examples to larger ones in the test set is exhibited.

Keywords: Learning to Act, Planning, Supervised Learning, Decision Lists, Structural Domains.

1 Introduction

In many problems of interest, an agent is required to choose actions in the world so as to achieve some goals. Such problems have been extensively studied in AI and largely in the subfield of

*A large part of this work was done while the author was at Harvard University and supported by ONR grant N00014-95-1-0550, and ARO grant DAAL03-92-G-0115.

planning. The general setup in planning assumes that the 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 [10, 18, 6]. On the positive side, sophisticated domain independent search methods that improve performance have been introduced (e.g. [48, 5, 20]). However the size of problems that can be solved, in terms of the number of objects in the problem, 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 have been constructed along this line [24, 27, 14, 47, 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 Explanation Based Learning [24, 27], static analysis [14], and analogy [45, 46] 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 [35] suggests to construct such a universal plan by planning for all contingencies and representing the result in some compact way.[1] Reactive plans are also related to Reinforcement Learning (RL). In this framework the agent learns by acting in the environment, generalising from the results of its own actions. Learning in RL is therefore only partly supervised. Reactive plans can be constructed by using dynamic programming to propagate values of states or actions [3]. This can be done in RL even when a model of the environment is not available, notably by using temporal difference methods [39, 19].

This paper reports on an application of supervised learning to problems of acting in the world, and has been motivated by previous theoretical results regarding such problems [23, 41]. In [23] a notion of learnability appropriate for (generalised 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 *strategy*[2] that can solve problems in the same domain. It is shown [23] that results on Occam algorithms generalise 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. That is, if a teacher uses a strategy in the class then a learner can find a strategy consistent with the 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 [23] is relaxed. In some of our experiments the examples are drawn by using a

[1] Some arguments were held regarding the utility of this approach; for some views and related technical results see [36, 16, 7, 37].

[2] A strategy is simply an algorithm that can be used to solve planning problems in a particular domain. In order not to confuse learning *algorithms* and *algorithms* for a planning domain we refer to the latter as strategies.

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 problem is NP-Complete [18]. This can be modelled as having "noise" in the examples and the learning algorithm must be able to tolerate such noise. The second issue addressed is that of efficiency. While the bounds in [23] are polynomial they are still rather high. We explore several practical issues that make this application possible. These include the use of action models by the learning algorithm, incorporating "background knowledge" in the form of additional predicates in the domain, the use of type information to prune the search space, and the use of the level-wise enumeration procedure recently introduced in data mining [1] in the process of enumerating 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 [18, 8, 37, 38] its structure is well understood so as to enable thorough analysis. The Logistics transportation domain [45] is more complex and has been recently studied from several perspectives [45, 47, 13, 20]. For each domain, random problems are drawn and presented with their solutions to the learning algorithm. The learning system uses a variant of Rivest's algorithm [33] to produce a PRS strategy represented as a first order decision list. 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 [5]) or by using a hand coded strategy for the domain.

As the experiments demonstrate, generalisation 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. As a result a search engine may be needed in case completeness is required. However, the learned strategies have some nice properties. Firstly, they are efficient — the running time is polynomial in the number of objects in the problem, where the degree of the polynomial is the number of variables in a rule. Secondly, the solutions produced by these strategies are not far from the optimal (shortest) solutions for the problems solved. Perhaps most importantly, the strategies can be used to solve some fraction of large planning problems — problems with domains that are larger than those given in the examples and in fact problems that are beyond the scope of domain independent techniques. These aspects are discussed quantitatively in Section 3. To summarise, the various experiments presented show that the learning algorithm is to some extent robust to noise, that it can be made efficient for small planning domains, and that the class of strategies is indeed expressive enough to handle such domains.

The rest of the paper is organised as follows. Section 2 describes the system, its interface, and the details of the learning algorithm. Section 3 describes the experimental setup and results. Section 4 briefly discusses related work and Section 5 concludes.

2 The System

The system, L2Act [22], includes a learning component and a performance evaluation component. The learning component receives as input a description of the domain and traces of solved problems, and produces a strategy for the domain represented as a set of rules. The performance evaluation 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 [5], 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 [22]. The description of the planning domain includes the names of predicates, and models of the actions given in a standard STRIPS [15] 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:[3]

$$\begin{aligned} \text{Base Rule:} \quad & on(x_1, x_2) \ \rightarrow above(x_1, x_2) \\ \text{Recursive Rule:} \quad & above(x_2, x_3)\ on(x_1, x_2) \rightarrow above(x_1, x_3) \end{aligned}$$

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 x_1, x_2, x_3 serve as object variables that can be bound to any object in the current state. This representation is the one used in [23]. It is straightforward to generalise this scheme to use any set of monotone rules that can be evaluated by repeated application of forward chaining.

Then, a set of runs (complete solved problems) follows. A run is composed of a set of objects, a set of propositions that hold in the start situation, a set of propositions that should hold in the goal and a sequence of actions that achieves the goal

2.2 Representation of Strategies

The learning algorithm produces as output an ordered list of existentially quantified rules. Following [23] 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 following rules:

$$\begin{aligned} \text{Rule:} \quad & OBJECT(x_1)\ OBJECT(x_2)\ clear(x_2)\ holding(x_1) \\ & G(on(x_1, x_2))\ G(\textit{on-table}(x_2))\ \textit{on-table}(x_2) \ \rightarrow STACK(x_1, x_2) \\ \text{Rule:} \quad & OBJECT(x_1)\ holding(x_1) \\ & G(on(x_1, x_2))\ \overline{clear(x_2)} \ \rightarrow \textit{PUT-DOWN}(x_1) \end{aligned}$$

As above, x_1 and x_2 are object variables. The predicate G is a marker for goal conditions so that $G(on(x_1, x_2))$ means that, in the goal, the block bound to x_1 should be on the block bound to x_2. The PRS is an ordered set of rules, and its semantics is as follows: As in decision lists [33], 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 current state 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 condition of a rule represents a conjunction of relational expressions. In the standard semantics of first order expressions two different variables are allowed to bind to the same object.

[3] The system actually uses an ASCII based representation for these rules; the details can be found in [22].

In some domains it can be useful to disallow this, namely, require that two object variables should not bind to the same object. For example, in the rules listed above this is the preferred interpretation though due to the semantics of the domain the added bindings will never satisfy the condition. In such a case, the efficiency of evaluating a rule is improved in the modified setting since useless bindings do not need to be checked. Our system includes a user controlled option to enforce this requirement and we used this option in all the experiments reported in the paper. Notice that this implicitly introduces inequalities on all variables and thus increases the expressiveness of the PRS. 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.

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 not too difficult to write algorithms that solve any problem and we have implemented such algorithms using PRS. The details are briefly discussed in Section 3. This addresses to some extent the issue of expressiveness of the class of strategies, showing applicability to planning domains of interest. The number predicates that are needed in the condition of the rules and the number of free variables used affect 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 [33] algorithm for learning decision lists. 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. Similarly, the PRS strategies use condition-action rules based only on the current state to choose an action.

The algorithm can be described independently of the particular set of rules under consideration. What we need is that the algorithm will be able to enumerate all possible rules and this must be a finite set. The system employs (user controlled) bounds on the number of predicates in the condition of the rules and number of variables in a rule to enable this enumeration. In its simplest form, a lexicographic enumeration of rules is used. We refer to this below as the *standard enumeration*.

A high level description of the algorithm is given in Figure 1. The algorithm runs in two phases. In the first phase the algorithm evaluates all the rules on all the examples. For each rule and each example it records two facts: 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 lexicographic order so that a rule can suggest only one action and the notion of correctness is well defined.

In the second phase the algorithm uses the information recorded to construct the PRS. This is done by starting with an empty list and repeatedly choosing the next rule on the list. In order to choose the next rule, the system uses a preference criterion that scores each rule according to the number of examples it covers and number of examples it covers correctly. A number of preference criteria are implemented in the system and these are discussed below. For any learning run, however, a single such criterion is fixed and used. The rule that maximises the preference score is chosen to be the next rule on the list, and all the examples covered by this rule are removed from the data set. This is repeated until all the examples have been covered and the data set is empty.

To complete the description of the algorithm we need to describe the preference criterion used in Step 3a. In addition, the system uses several techniques to improve efficiency by not enumerating

1. Enumerate all rules under consideration
   (a) Enumerate all examples in the data set
      i. Enumerate all possible bindings (of variables in the current rule to objects in the current example)
         A. If the condition of the current rule is satisfied by the current example under the current binding then
            - Mark that the rule covers the example
            - Test and mark whether the rule is correct on the example
            - Continue to the next example (that is, quit the binding enumeration loop)
2. Initialise the PRS to the empty list
3. While the data set is not empty:
   (a) Choose the most preferable rule according to the preference criterion
   (b) Add it to the end of the PRS
   (c) Remove all examples that are covered by this rule from the data set

Figure 1: The Learning Algorithm.

irrelevant rules and bindings in the first phase. These aspects are described in the next two subsections.

2.3.1 Preference Criteria

As in the case of propositional decision lists [33] 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. By repeatedly picking such a rule in each iteration we are guaranteed to find a consistent PRS.

In case the input is not produced by a PRS, the situation is less clear. Let *cover* be the number of examples covered by a rule and *correct* the number of examples on which it is correct. The point in question is exemplified by the following: suppose one rule has $cover = 1$ and $correct = 1$ and another has $cover = 88$ and $correct = 89$. Which one should we choose? The first rule does not introduce any errors on the data set but we do not have enough evidence to be sure about it. The second one introduces some error but we can be quite sure that it is a small error. We have experimented with several preference criteria as follows.

> **PF0**: Prefer rules with higher $correct/cover$ ratio, and in case of a tie prefer the rule that covers more examples.

The criterion PF0 is the default criterion in the system and unless otherwise specified all experiments reported use it. Notice that in the above example it would prefer 1/1 to 88/89. With PF0 the algorithm coincides with Rivest's if there is always a rule consistent with the examples. It is therefore guaranteed to succeed in such a case.

For the next criterion let the initial data set size be N. Note that *cover* and *correct* are updated in each iteration. Namely, they may be reduced if an example covered by the rule was removed from the data set.

> **PF1**: Fix a small constant $0 < \alpha < 1$ (close to 0). If there is any rule such that $cover/N < \alpha$ then consider only rules with this property. Otherwise consider all rules. In both cases, of the rules considered, choose as in PF0, that is: prefer rules with higher $correct/cover$ ratio, and in case of a tie prefer the rule that covers more examples.

The criterion PF1 is suggested by [21] who show that the algorithm using it can tolerate random classification noise in the probably approximately correct (PAC) learning model. Intuitively, this is based on the assumption that the set of examples is large enough so that the statistics for any rule in any position in the list is reasonably accurate. In such a case, if α is small then a rule that covers only an α fraction of the examples cannot do much harm. On the other hand when no such rule exists we are guaranteed to find a rule of high accuracy. In order for this analysis to apply α must be set to a value that depends on N and the error parameters of the algorithm. For our setting this requires a large N and very small α so that it could not be applied directly. In our experiments we arbitrarily set $\alpha = 0.01$.

> **PF2**: Fix a constant $0 < \gamma < 1$ (close to 1). If there is any rule such that $cover/correct > \gamma$ then consider only rules with this property. Of these rules prefer the one that covers the largest number of examples.
>
> Otherwise consider all rules and choose as in PF0, that is: prefer rules with higher $correct/cover$ ratio, and in case of a tie prefer the rule that covers more examples.

The intuition behind PF2 is that rules that cover a large number of examples should be admitted even if they introduce a small error. Since we can expect that some error will be introduced in any case, the greedy nature of the algorithm suggests that covering the examples in larger chunks will produce smaller PRS with hopefully reduced total error. In particular, in the example above PF2 would prefer 88/89 to 1/1. In our experiments we arbitrarily set $\gamma = 0.9$.

2.3.2 Reducing Complexity

In the discussion of the algorithm we have ignored the source or structure of the rules. Recall the rule structure mentioned above:

$$\begin{aligned} \text{Rule:} \quad & OBJECT(x_1)\ OBJECT(x_2)\ clear(x_2)\ holding(x_1) \\ & G(on(x_1, x_2))\ G(on\text{-}table(x_2))\ on\text{-}table(x_2)\ \rightarrow STACK(x_1, x_2) \end{aligned}$$

One can quantify the number of rules 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 $(4n)^{k_R} k_B^{k_R k_A}$ (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 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 $(4n)^{k_R} k_O^{k_R k_A}$. By fixing k_B to a small constant we in fact reduce the number of rules and the size of hypothesis class (assuming of course that we are interested in solving problems with large k_O). 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 sample 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, for example, by not enumerating self-contradictory conditions that are never satisfied. For the problem of enumerating bindings, if we find one binding that fails to satisfy the condition, we can prune other bindings that will fail for the same reason. These and several other techniques are used in the system; we discuss a few that are particularly relevant for the problem of planning.

Using Action Models

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 $STACK$. If searching for this rule from scratch we need 7 predicates in the condition. By fixing the first 4 to include the preconditions we only need to find the remaining 3. Thus, action models help in focusing the learner in its search for good rules by simply appending the preconditions as part of the rules. Clearly, an implementation of this is straightforward.

Identifying Goal Predicates

The other techniques we use utilise information in the examples to restrict the enumeration of rules. A trivial application can be used for goal predicates. Since in principle each predicate can appear with a goal modality but in practice only some do, an initial search can reduce the number of candidate predicates. We scan the example runs and mark all the predicates that ever appeared as part of the goal. Only these are used with a goal marker in the construction of rules.

Using Type Information

A more dramatic application of this idea can be seen in domains where types play an important role. For example in the Logistics domain there are objects of several types including $OBJECT$, $TRUCK$, $AIRPLANE$, $CITY$, as well as the predicate $in(x, y)$. By scanning the example runs we observe that the predicate $in()$ accepts only $OBJECT$ in the first parameter and only $TRUCK$ or $AIRPLANE$ in the second parameter. Now clearly one should not try to use a rule with a construct like $AIRPLANE(x)CITY(y)in(y, x)$.

Again the implementation is straightforward. The type predicates are identified as part of the input to the system. The example runs are scanned and each argument of each predicate is marked with the set of types that it accepts. The rules generated are restricted so that every variable must have at least one type common to all its occurrences. In addition, since the type restrictions are automatically included in the rules we do not need to include them explicitly in the condition. This is easily achieved and reduces the number of rules considerably. Moreover, with a bit of book-keeping this also helps in the problem of binding enumeration, since unreasonable bindings do not need to be considered. This can be done by computing the set of types allowed for each variable in a rule. Given a new example we can compute the set of objects that may bind to each variable and consider only these. We note that in the blocks world there is only one type of objects and thus the technique is not applicable.

Level-wise Enumeration

Another application of this general idea follows Valiant's [42] suggestion for learning DNF expressions. There, one first enumerates conjunctions that appear in the examples with some minimal frequency, and only then tries to learn a disjunction of these. 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 proved useful in practice and is known to be optimal in some special cases [1, 26, 17]. Following [26], we refer to the algorithm as the *level-wise algorithm*. The basic observation is that if a condition covers a large fraction of the examples then all subsets of the condition cover at least the same fraction of the examples. Therefore, conditions that are satisfied frequently can be found using a bottom up search, each time combining frequent sub-conditions into longer ones. A version of this algorithm adapted to the problem of learning PRS is used in the system. This (user controlled) option replaces the first phase of the learning algorithm by the following procedure.

Fix a frequency threshold $\sigma < 1$ (say $\sigma = 0.01$). Let N be the size of the data set and *cover* the number of examples covered by a rule. In the following by the size of a rule or size of a condition we mean the number of predicates in the condition.

The procedure works in iterations, where in the i'th iteration it (1) constructs a set of candidate rules of size i, (2) evaluates each of these rules on the examples, and (3) removes any rule with $cover/N < \sigma$ from the set of candidates.

Step (1) above is done as follows: In the first iteration the construction of candidates simply amounts to a lexicographical enumeration of all rules with a condition of size one. For $i > 1$, we construct rules with condition of size i from ones with condition of size $i-1$. This can be done as follows. (1.1) Find two rules of size $i-1$ that have identical parts of size $i-2$. (1.2) Construct a rule of size i from these by joining their conditions. (1.3) Check that all subsets of the new condition appear as frequent rules in level $i-1$. By using an appropriate representation for the candidate rules one can implement these operations efficiently.

This technique adds one more parameter to the setup of the system, namely the frequency threshold used; below we refer to this parameter as σ. Naturally, the threshold should depend on the average solution length. For example, a rule that is used once in every solution should be included in the enumeration.

Taken together, these techniques reduce the number of rules by several orders of magnitude as well as reducing binding time. The complexity is however still somewhat high. We have used the system for the blocks world with $k_R = 2$ and $k_B = 3$. For the logistics domain, the additional type checking allows us to use $k_R = 3$ 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 $k_B = 6$.

2.4 Performance Component

The system also includes a performance component that gets as input a description of the domain, a PRS, and a set of runs. The system 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 performance component can also be used as a programming environment for PRS and can therefore help debug hand-coded PRS.

3 Experimental Results

We have experimented with two domains. Both were studied before and were used to generate challenge problems for various systems. Due to recent studies [18, 8, 37, 38] the structure of the blocks world domain is well understood so as to enable thorough analysis. The Logistics transportation domain [45] is more complex, including more predicates and operators, larger arity for operators, and objects of various types.

3.1 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 includes: $PICKUP(x)$, $PUTDOWN(x)$, $UNSTACK(x,y)$, $STACK(x,y)$ with the obvious semantics. The predicates are $on(x,y)$, $clear(x)$, $on\text{-}table(x)$, $holding(x)$, $OBJECT(x)$. 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 (for example, the position of block 2 may be unspecified).[4] 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 [18]. 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.

3.1.1 Experimental Setup

Our learning algorithm receives as examples problems descriptions and solutions for them. We generate examples in the following manner (1) Random blocks world states are generated using *bwstates* a program written by Slaney and Thiebaux [38]. (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 [5] is used to solve the problems.

We also supply the algorithm with knowledge about the domain in the form of support predicates, as explained in the previous section. In particular the following predicates were given:

$$\begin{aligned}
\text{Base Rule:} \quad & G(on(x_1,x_2)) \rightarrow ingoal(x_1) \\
\text{Recursive Rule:} \quad & G(on\text{-}table(x_1)) \rightarrow ingoal(x_1)
\end{aligned}$$

$$\begin{aligned}
\text{Base Rule:} \quad & G(on\text{-}table(x_1))\ on\text{-}table(x_1) \rightarrow inplacea(x_1) \\
\text{Recursive Rule:} \quad & inplacea(x_2)\ G(on(x_1,x_2))\ on(x_1,x_2) \rightarrow inplacea(x_1)
\end{aligned}$$

$$\begin{aligned}
\text{Base Rule:} \quad & G(on(x_1,x_2))\ on(x_1,x_2)\ \overline{ingoal(x_2)} \rightarrow inplaceb(x_1) \\
\text{Recursive Rule:} \quad & inplaceb(x_2)\ G(on(x_1,x_2))\ on(x_1,x_2) \rightarrow inplaceb(x_1)
\end{aligned}$$

[4] The fact that the goal is only partially specified makes the problem a bit more tricky due to the following: A partial stack of blocks may have all its goal conditions satisfied while being above a block that must be moved since it belongs to a different stack in the goal. In such a case all these blocks must be moved.

$$\begin{aligned} \text{Base Rule:} \quad & on(x_1, x_2) \ \rightarrow above(x_1, x_2) \\ \text{Recursive Rule:} \quad & above(x_2, x_3)\ on(x_1, x_2) \ \rightarrow above(x_1, x_3) \end{aligned}$$

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 $k_B = 3$. We generated 20 independent data sets each of size 4800 examples and ran the learning algorithm on each of these. Note that by example we mean a situation-action pair. The number of complete problems corresponding to the 4800 situation action pairs was roughly 315. Unless otherwise specified all experiments used the level-wise algorithm described in Section 2.3.2 with frequency threshold $\sigma = 0.01$, and the preference criterion PF0 described in Section 2.3.1. The learning time for these experiments was roughly 13 minutes (for training with 4800 examples each with 8 blocks), on a PC using a Pentium 2/400MHz processor. The learning time grows roughly linearly with the number of examples since the dominating factor is the time to evaluate the rules.

In order to test the strategies produced by the learning algorithm we generated random test problems, with 7, 8, 10, 12, 15, and 20 blocks. For each size we used 1000 planning problems. A strategy succeeds on an example if it solves the planning problem (achieves the goal); no partial credit was given otherwise. In all the graphs plotted below, every point represents 20,000 attempts to solve planning problems, from 20 runs of the learning algorithm each tested against 1000 problems of a particular size. We therefore get good estimates for the error of any strategy tested (the standard deviation for estimating the probability of success in 1000 Bernoulli trials is bounded by 0.016). However, since we made only 20 runs of the algorithm the estimates of the expectation of the error is less good. Following [28], if we assume that the error estimate of each individual run has a normal distribution then we can use a t confidence interval for each point in our graphs.[5] Instead of drawing these we note here that the the largest 99% confidence interval in any of the graphs plotting success rate is ± 0.10.

3.1.2 Results

Figure 2 describes the fraction of problems solved by the output of the learning algorithm as a function of number of examples, for several problem sizes. As one can observe generalisation is achieved, and a significant fraction of the problems is solved by the strategies. In particular, 79 percent of problems of the same size (8 blocks), and 48 percent of large problems (with 20 blocks) are solved. For comparison, we ran GraphPlan on 10 of the test problems with with 12 blocks and only one was solved in less than half an hour. We can also see that the performance stops improving well before using the entire sample. Namely, the sample size is sufficiently large.

Figure 3 shows how the success rate scales with the size of the problems (for learning with 4800 examples) and several preference criteria. The line marked PF0 corresponds to the same experiments as in Figure 2. 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.

[5] This relies on the fact that each runs measures a sum of many random variables. The assumption is not fully justified here since the variables are correlated. The derivation for the t statistic can be found in [4]. The number of repetitions is $n = 20$ and the t quantile $t_{19}(0.995) = 2.861$. Let v_i the estimate we get from each repetition, then for confidence 99% we need $\pm 2.861 \sqrt{\frac{1}{380} \sum (v_i - \overline{v})^2}$.

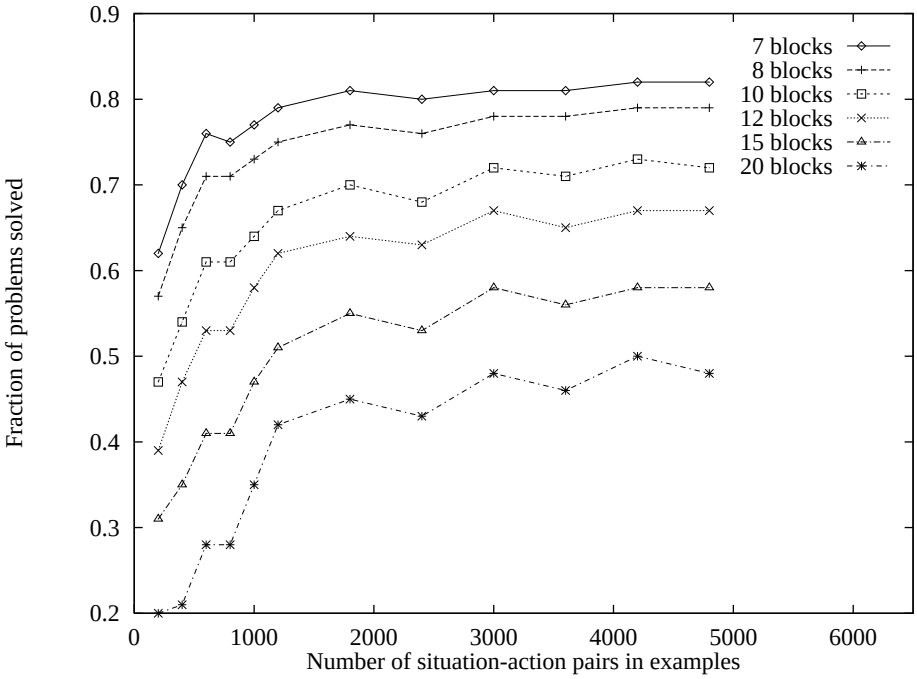

Figure 2: Success rate of learned strategies.

Several further points are worth noting here. First, while we did not plot the variation, the differences between runs are substantial. For example, of the 20 experiments averaged the best run produced strategies that solved 89 percent of problems with 8 blocks, and 79 percent of problems with 20 blocks. Second, the PRS strategies are efficient; the time for solving a problem with 20 blocks is roughly one second. The strategies do get slower for larger problem sizes, however the time required grows polynomially, where the dominant factor is the matching time $k_O^{k_B}$. Finally, some fraction of problems remains unsolved by the learned PRS. In this respect note that we a 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.

3.1.3 Preference Criteria

As mentioned above we have experimented with several preference criteria for choosing between rules. Recall from Section 2.3.1 that we defined three preference criteria identified as PF0, PF1, PF2. Figure 3 plots the success rate as a function of the number of blocks for the three criteria (for learning with 4800 examples). Using the same argument as above we can get a t-test to identify differences between the criteria by using the difference in success rate as the estimated quantity. This identifies a difference for a gap of at least 0.10 so that some of these differences are accepted as significant. More importantly, however, these experiments and their confidence intervals support the claim that all these criteria induce strategies that solve a non-negligible fraction of problems, and that transfer to problem of larger size does occur.

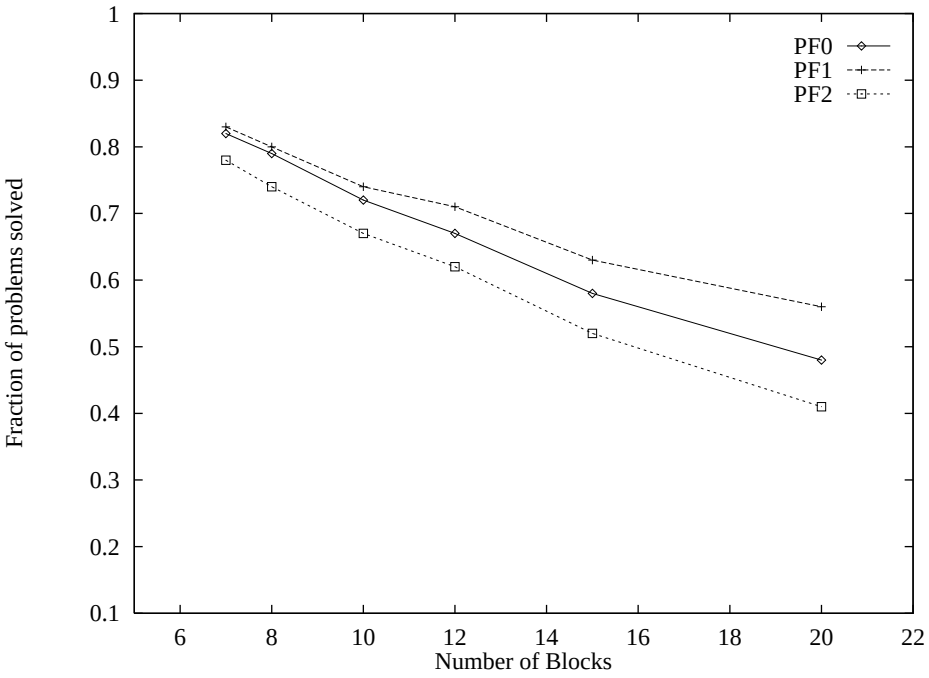

Figure 3: Success rate as a function of the number of blocks.

3.1.4 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 [38] 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 as a function of the number of blocks (for learning with 4800 examples). As can be seen the learned strategies perform better than these algorithms. The ratio against GN1 is 0.98 for 8 blocks and 0.94 for 20 blocks. The ratio against US is 0.91 for 8 blocks and 0.86 for 20 blocks. Figure 5 concentrates on GN1 plotting the ratio of solution lengths as a function of the number of examples. It can be seen that the average behaviour is stable and that for larger problems the difference between the learned strategies and GN1 becomes more pronounced. Interpolating from the graphs in [38] we see that the ratio of optimal solution length to GN1 is 0.94 for 20 blocks.

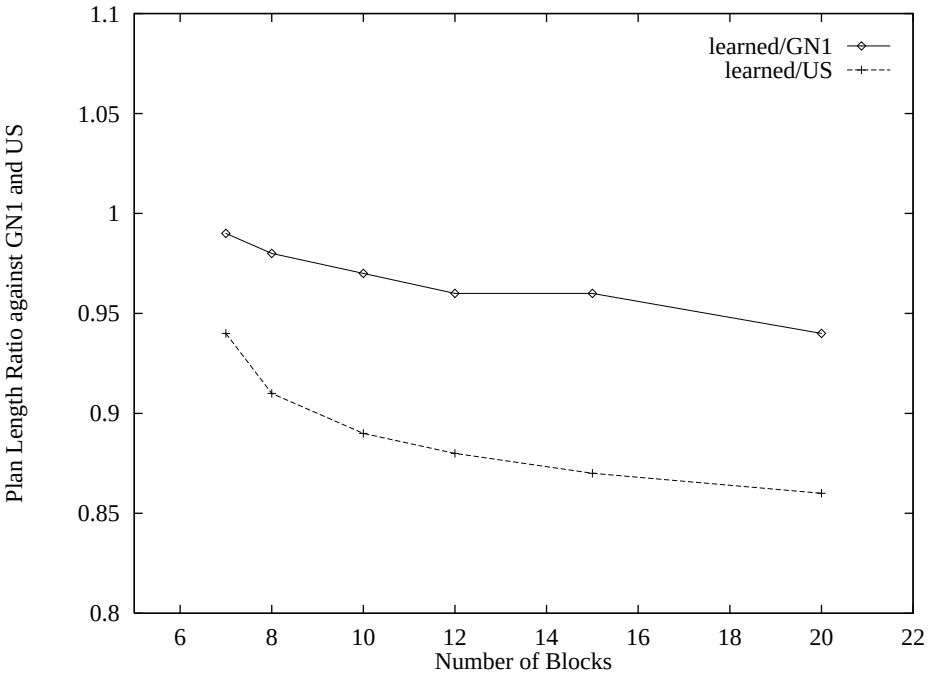

Figure 4: Ratio of solution length: learned strategies compared with US, and GN1.

The 99% confidence interval for our estimate of the quality is ± 0.01. We therefore conclude that learned strategies produce solutions of high quality.

3.1.5 Threshold for Enumeration

All the experiments reported above used the bottom up level-wise algorithm in enumerating the candidate rules, with frequency threshold $\sigma = 0.01$. Namely a rule was considered in the learning algorithm only if it covered at least one percent of the original sample. Clearly the threshold can affect 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.

Figure 6 plots the performance of learned strategies for several value of σ. Also plotted is the performance of the algorithm when using the standard lexicographic enumeration of rules (with no cutoff threshold). One can see that $\sigma = 0.05$ produces much worse results but that other values of σ are close to the standard enumeration. That is, the gain in efficiency for these values does not come at a cost in performance.

The reduction in the number of rules used in the learning algorithm and hence running time is less regular than one might expect. Figure 7 plots the performance, as well as the number of rules enumerated as a function of σ (the number of rules is normalised where 1 corresponds to 3390). The standard enumeration is included as $\sigma = 0$ in this graph. 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 $\sigma = 0.05$. When using the system in level-wise mode care must be taken to use an appropriate value of σ. The above characterisation may help in finding such a value.

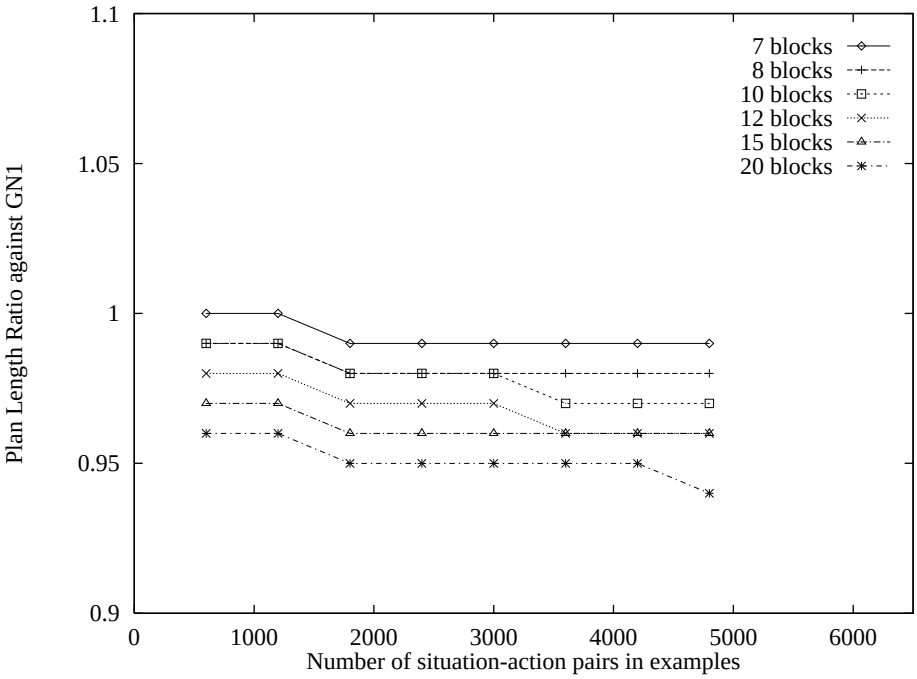

Figure 5: Ratio of solution length: learned strategies compared with GN1.

It is interesting to study the reason for the slow decrease in the number of rules after the initial step. A possible 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. Experiments in which such spurious rules are filtered are an interesting direction for future work.

3.2 The Logistics Domain

The Logistics transportation domain introduced by Veloso [45] is more complex, including more predicates and operators, larger arity for operators, and objects of various types. The domain describes a simplified scheduling problem for a company shipping packages using trucks and airplanes. The setup includes several cities, in each city several locations and some locations are designated as airports. Within a city one can ship packages using trucks but between cities one must use an airplane. The planning problem is: given a set of packages in various locations, their destination locations, and the current locations of trucks and airplanes, schedule the use of trucks and planes so as to deliver the packages. The domain is an abstraction of a real transportation domain and has been recently studied in several frameworks [20, 13, 46, 47].

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 $at(x, y)$ indicating the location of objects and vehicles, $in(x, y)$ for indicating that some objects are in some vehicle, and $loc\text{-}at(x, y)$ to indicate in which city a location resides. The domain also includes the actions $LOAD\text{-}TRUCK$, $LOAD\text{-}AIRPLANE$, $UNLOAD\text{-}TRUCK$, $UNLOAD\text{-}AIRPLANE$, $DRIVE\text{-}TRUCK$, $FLY\text{-}AIRPLANE$ with the expected definitions. A complete description can

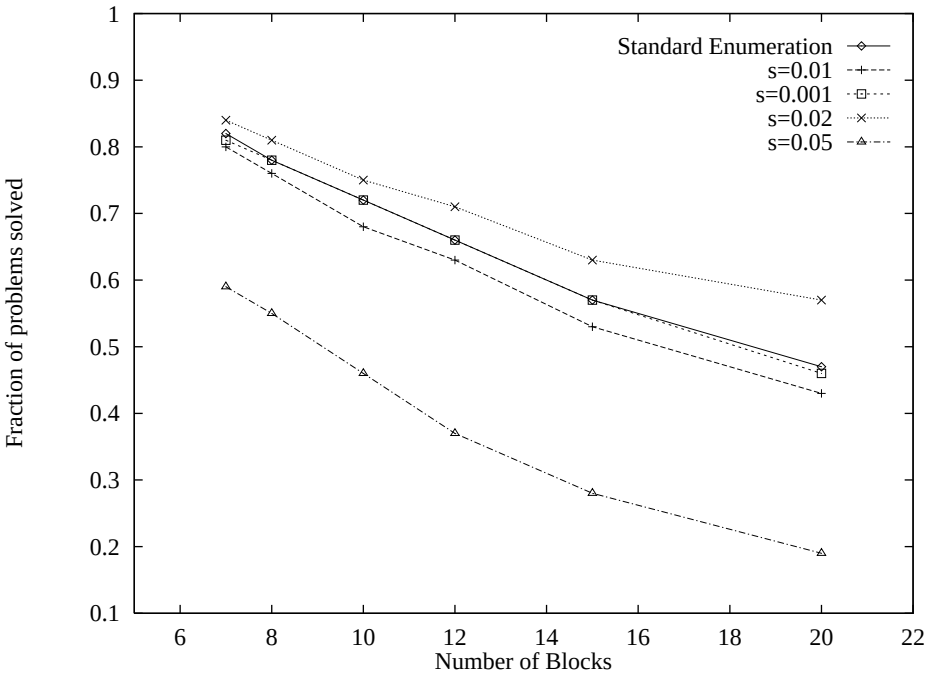

Figure 6: The effect of threshold on performance.

be found in [22].

3.2.1 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. These parameters were chosen in an attempt to use the smallest problems that include "sufficient information" though this was not tested in a rigorous way.

Preliminary experiments using GraphPlan [5] 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 solutions using a PRS that was hand coded. The PRS encodes simple conditions for unloading trucks and airplanes, loading them if packages need to be moved, and driving and flying if they are loaded and their packages need to go elsewhere. By prioritising the rules in the order described we avoid loops. This PRS has $k_R = 4$ and $k_B = 6$ and it produces solutions of comparable length to those of GraphPlan.

Using the above method we generated example problems, all of which included 2 packages, and trained the algorithm 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 generated 20 independent data sets each of size 1200 examples and ran the learning algorithm on each of these. Note that by example we mean a situation action pair. The number of complete problems corresponding to the 1200 situation action pairs was roughly 115. All experiments for this domain

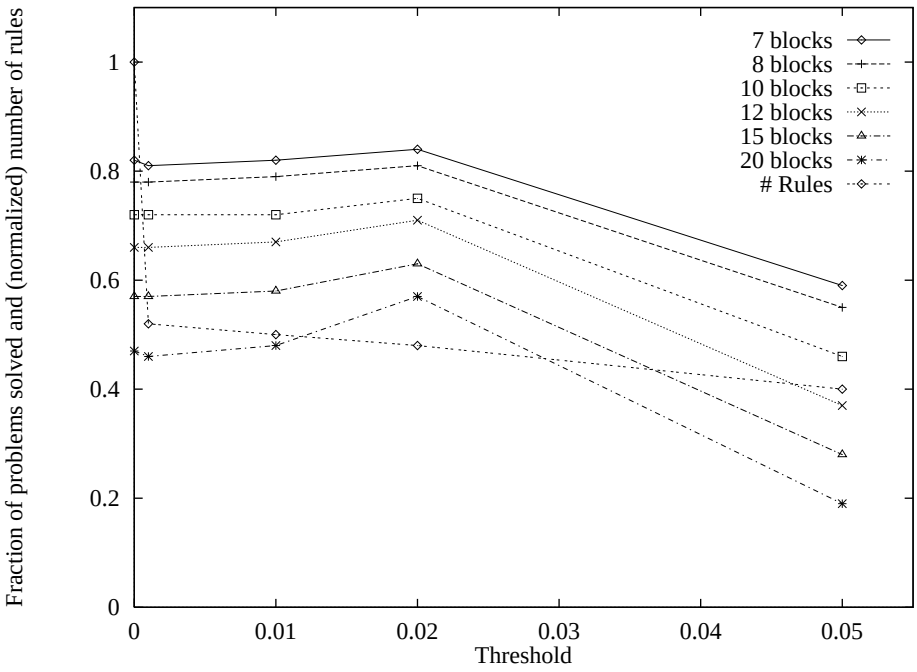

Figure 7: Performance and number of rules as a function of threshold.

used the level-wise algorithm described in Section 2.3.2 with frequency threshold $\sigma = 0.01$. The learning time for these experiments was roughly 150 minutes (for training with 1200 examples each with 2 packages and a total of 16 objects), on a PC using a Pentium 2/400MHz processor. As before, the learning time grows roughly linearly with the number of examples.

In order to test the strategies produced by the learning algorithm we generated random test problems with 2, 6, 10, 15, 20, and 30 packages. For each size we used 1000 planning problems so that the statistical consideration as the same as in the blocks world experiments.

3.2.2 Results

Figure 8 plots the success rate of the learned strategies on the various test problem sizes, for the 3 preference criteria PF0, PF1, PF2 as discussed above. The 99% confidence intervals are of size $\pm 0.10, \pm 0.11, \pm 0.15$ respectively. The experiments confirm the hypothesis that all three criteria lead to strategies that solve a non-negligible fraction of problems, and that transfer to problem of larger size does occur.

Here again for comparison we ran GraphPlan on 10 of the test problems with 20 packages, and none was solved in less than half an hour. In contrast for the problems solved of this size the PRS took less than 3 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 the number of cities and locations is too small in our experiments. As a result, with many packages it is likely that one has to visit all locations in order to ship everything, and simple conditions are easily identifiable making the problem easier for reactive strategies. In order to test this hypothesis we drew another set of test problems, 200 of each size, with 4 cities and same characteristics as before (so that there are 2 more locations and 1 more truck). The lines marked "PF0 - 4 cities" in Figure 8 describes the

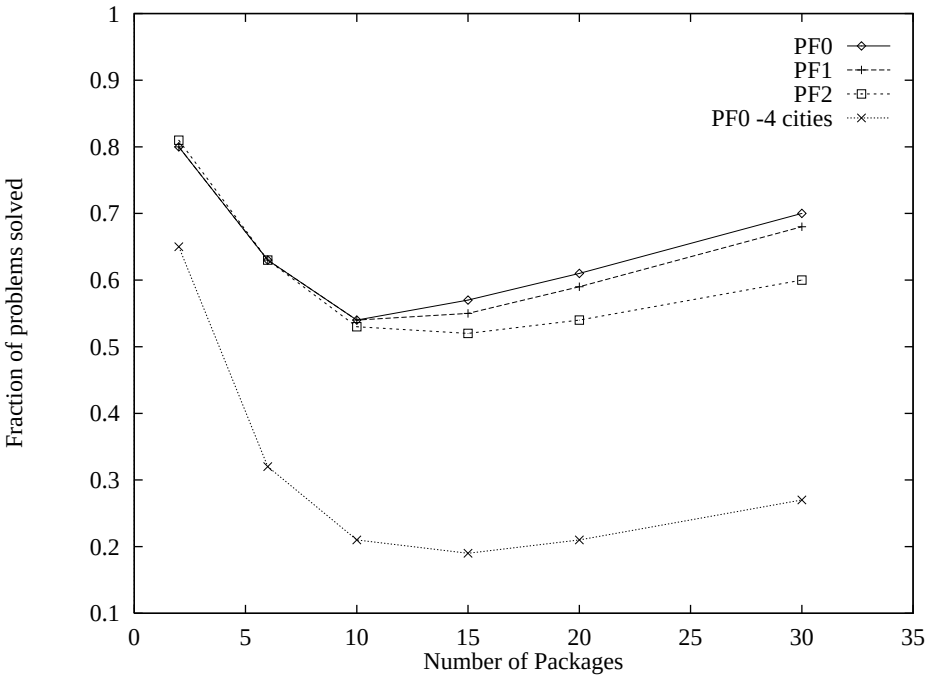

Figure 8: Success rate of learned strategies in the Logistics domain.

performance of strategies learned using the PF0 criterion on this set. Indeed the minimum in these tests shifts to right though the differences are not statistically significant. The plot demonstrates that we also get generalisation in terms of the number of cities and locations. However, it also shows that the performance deceases considerably when all parameters are varied. More work is needed to identify a size for training problems enabling good generalisation in all dimensions.

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.[6] We conclude that successful strategies that produce good solutions are learned.

Blocks World	7	8	10	12	15	20
	0.83	0.80	0.74	0.71	0.63	0.56
Logistics	2	6	10	15	20	30
	0.80	0.63	0.54	0.55	0.59	0.68

Figure 9: Ratio of problems solved as a function of problem size (PF1 criterion).

[6]It may be worth noting here that GraphPlan finds a "shortest parallel solution" requiring a minimum number of stages where independent actions can happen in parallel. This does not always yield the shortest sequential solution.

3.3 Summary

Figure 9 summarises the performance achieved in our experiments in terms of the average ratio of problems solved by learned strategies. The results given are for the level-wise algorithm with $\sigma = 0.01$. Since the criterion PF1 gives the best combined results we include the data for PF1 in this summary. When using the system one should adjust these parameters to fit the problem. Figure 7 shows a tradeoff in using σ that may help in doing so.

4 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 Explanation Based Learning methods [11, 29] used in Prodigy [27] and in Soar [24, 34], static analysis in Prodigy [14], analogy in Prodigy [45, 46, 47], and Inductive Logic Programming in Scope [13]. 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. In this area our model is closest to several works by Tadepalli et. al. [40, 41, 32] 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 of the assumption on existence of consistent strategies. Our work is also related to Reinforcement Learning [19] and to Inductive Logic Programming [30]. An extensive discussion comparing our approach with related work appears in [23]. Below we briefly discuss work concerning the planning domains.

While several 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 [27, 14, 45]. This set typically includes problems of various sizes, some of them of relatively small size. For example, for the blocks world, Minton [27] (and following him other works in Prodigy) reports on one set of 100 problems with 3 to 12 blocks. Estlin and Mooney [13] 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 [13] report on problems with 2 packages. Veloso *et. al.* [47] 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 here (success rate of 59% on 20 packages), though the the evaluation procedure was somewhat different.[7] 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.

[7] Success rate for this particular instance is reported there under a time bound of 450 seconds in their system.

5 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.

To summarise, the contribution of the paper is in applying a theoretically justified algorithm to learn first order rule based strategies for problem solving. In doing so we demonstrated that the algorithm can tolerate noise and devised various techniques for improving its efficiency in practice. We have also partly validated a claim for expressiveness by coding PRS strategies for both domains.

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. [12] for recent work) and its improvement can affect 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 [9] and FOIL [31] use a similar representation and can be applied. Another direction is to apply our approach to stochastic domains studied in Reinforcement Learning.

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. Valiant [43, 44] suggests that action strategies can be embedded in the Neuroidal architecture thus taking advantage of the robustness of threshold elements and their algorithms, particularly the Winnow algorithm [25] that is useful for handling irrelevant attributes. Preliminary experiments using a modification of this algorithm with our system did not yield good performance, and more work is needed to investigate this direction.

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.

Acknowledgements

I am grateful to Les Valiant for many helpful discussions and suggestions, and to Prasad Tadepalli, Bart Selman, and the anonymous referees for their comments. I am also grateful to John Slaney and Sylvie Thiebaux for the use of their program bwstates, and Avrim Blum and Merrick Furst for the use of their program GraphPlan.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 207 – 216, Washington, DC, May 1993. ACM Press.

[2] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proceedings of the 3rd European Workshop on Planning*, 1995.

[3] A. G. Barto, S. J. Bradtke, and A. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[4] P. Bickel and K. Doksum. *Mathematical Statistics*. Prentice Hall, 1977.

[5] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the International Joint Conference of Artificial Intelligence*, pages 1636–1642, August 1995.

[6] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[7] D. Chapman. Penguins can make cake. *AI Magazine*, 10(4):45–50, 1989.

[8] S.V. Chenoweth. On the NP-harness of blocks world. In *Proceedings of the National Conference on Artificial Intelligence*, pages 623–628, 1991.

[9] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[10] S. A. Cook. The complexity of theorem proving procedures. In *3rd annual ACM Symposium of the Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971. ACM Press.

[11] G. DeJong and R. Mooney. Explanation based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.

[12] R. B. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the National Conference on Artificial Intelligence*, pages 290–296, 1993.

[13] T. A. Estlin and R. J. Mooney. Multi-strategy learning of search control for partial-order planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 843–848, 1996.

[14] O. Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62:255–301, 1993.

[15] R. E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5, 1971.

[16] M. Ginsberg. Universal planning: An (almost) universally bad idea. *AI Magazine*, 10(4):40–44, 1989.

[17] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proceedings of the symposium on Principles of Database Systems*, pages 209–216, 1997.

[18] N. Gupta and D.S. Nau. Complexity results for blocks world planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 629–633, Anaheim, CA, 1991. AAAI Press.

[19] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[20] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[21] M. J. Kearns and R. E. Schapire. Efficient distribution-free learning of probabilistic concepts. *Journal of Computer and System Sciences*, 48:464–497, 1994.

[22] R. Khardon. L2Act: User manual. Technical Report TR-10-97, Aiken Computation Lab., Harvard University, May 1997.

[23] R. Khardon. Learning to take actions. *Machine Learning*, 35(1):57–90, 1999.

[24] J.E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[25] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[26] H. Mannila and H. Toivonen. On an algorithm for finding all interesting sentences. In *Cybernetics and Systems, The Thirteenth European Meeting on Cybernetics and Systems Research*, pages 973 – 978, 1996.

[27] S. Minton. Quantitative results concerning the utility of explanation based learning. *Artificial Intelligence*, 42:363–391, 1990.

[28] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[29] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation based learning: A unifying view. *Machine Learning*, 1:47–80, 1986.

[30] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 20:629–679, 1994.

[31] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[32] C. Reddy, P. Tadepalli, and S. Roncagliolo. Theory guided empirical speedup learning of goal decomposition rules. In *International Conference on Machine Learning*, pages 409–416, Bari, Italy, 1996. Morgan Kaufmann.

[33] R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[34] P. S. Rosenbloom, J. E. Laird, and A. Newell. *The Soar papers : Research on integrated intelligence*. MIT Press, Cambridge, MA, 1993.

[35] M. Schoppers. Universal plans for reactive robots in unpredictable domains. In *Proceedings of the International Joint Conference of Artificial Intelligence*, pages 1039–1046, Milan, Italy, 1987. Morgan Kaufmann.

[36] M. Schoppers. In defense of reaction plans as caches. *AI Magazine*, 10(4):51–62, 1989.

[37] B. Selman. Near-optimal plans, tractability, and reactivity. In *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 521–529, Bonn, Germany, 1994. Morgan Kaufmann.

[38] J. Slaney and S. Thiebaux. Linear time near-optimal planning in the blocks world. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1208–1214, Portland, Oregon, 1996. AAAI Press.

[39] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

[40] P. Tadepalli. A formalization of explanation based macro-operator learning. In *Proceedings of the International Joint Conference of Artificial Intelligence*, pages 616–622, Sydney, Australia, 1991. Morgan Kaufmann.

[41] P. Tadepalli and B. Natarajan. A formal framework for speedup learning from problems and solutions. *Journal of Artificial Intelligence Research*, 4:445–475, 1996.

[42] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of the International Joint Conference of Artificial Intelligence*, pages 560–566, Los Angeles, CA, 1985. Morgan Kaufmann.

[43] L. G. Valiant. Managing complexity in neuroidal circuits. In *Proceedings of the 7th International Workshop on Algorithmic Learning Theory*. Springer-verlag, 1996.

[44] L. G. Valiant. Neuroidal architecture for cognitive computation. In *ICALP'98, The International Colloquium on Automata, Languages, and Programming*, pages 642–669. Springer-verlag, 1998. Lecture notes in Computer Science, vol. 1443.

[45] M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. Also appeared as technical report CMU-CS-92-174.

[46] M. Veloso. Flexible strategy learning: Analogical replay of problem solving episodes. In *Proceedings of the National Conference on Artificial Intelligence*, pages 595–600, 1994.

[47] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating learning and planning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[48] D. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.