Learning generalized policies in planning using concept languages

Mario Martín
Dept. de Llenguatges i Sistemes Inf.
Universitat Politècnica de Catalunya
08034 Barcelona, Spain

Héctor Geffner
Departamento de Computación y TI
Universidad Simón Bolívar
Aptdo. 89000, Caracas, Venezuela

Abstract

In this paper we are concerned with the problem of learning how to solve planning problems in one domain given a number of solved instances. This problem is formulated as the problem of inferring a function that operates over all instances in the domain and maps states and goals into actions. We call such functions *generalized policies* and the question that we address is how to learn suitable representations of generalized policies from data. This question has been addressed recently by Roni Khardon [16]. Khardon represents generalized policies using an ordered list of existentially quantified rules that are inferred from a training set using a version of Rivest's learning algorithm [22]. Here, we follow Khardon's approach but represent generalized policies in a different way using a *concept language*. We show through a number of experiments in the blocks-world that the concept language yields a better policy using a smaller set of examples and no background knowledge. The policy representation is related to the indexical-functional representations advocated by Agre and Chapman [1] and domain concepts such as 'the-next-needed-block' and 'a-well-placed-block' are identified from scratch.

1 Introduction

Planning is an essential part of intelligent behavior. In AI, a planner is given an initial state and a goal, and finds a sequence of actions that maps the state into the goal [23]. This problem has been tackled by a number of algorithms and in recent years substantial progress has been made [6, 15]. Still the problem is computationally hard and the best algorithms are bound to fail on certain classes of instances [10]. An alternative that has been proposed is to use knowledge of the planning domain for guiding the planning process (e.g., [3]). In the blocks world, for example, one may want to say 'never to move a block A that is well-placed' where the predicate 'well-placed' is defined in a suitable way. Planners that rely on domain-dependent control knowledge can outperform the best domain-independent planners [3, 11] but such knowledge is not always easy to provide.

In this paper, we are concerned with the problem of *learning* domain-dependent control knowledge. More precisely, we will be interested in the problem of learning how to solve a problem in a domain, given solutions to a number of small instances. This problem is formulated as the problem of inferring a function that operates over all instances in the domain and maps states and goals into actions. We call such functions *generalized policies* in contrast with the policies used in dynamic programming that have a more limited scope. A generalized policy for the blocks-world may say things like 'pick up a misplaced block if clear', 'put current block on destination if destination block is clear and well placed', etc. The question that we address is how to learn suitable representations of such policies from a number of solved instances. This question has been addressed recently by Roni Khardon [16]. Khardon represents generalized policies using an ordered list of existentially quantified rules and infers such representations from data using a version of Rivest's learning algorithm [22]. Khardon's results are encouraging and show that the learned policies can solve problems that the planner used as teacher cannot solve. The approach, however, has also some weaknesses. First, it relies on background knowledge in the form of support predicates that express key features of the domain, and second, the resulting policies do not generalize so well. In this paper we aim to show that these weaknesses can be addressed by learning generalized policies expressed using a *concept language* [8, 9, 12, 19]. Concept languages have the expressive power of fragments of standard first-order logic but with a *syntax* that is suited for representing and reasoning with *classes* of objects. For example, the class of 'well-placed' objects in the blocks world domain can be defined in a very compact way in terms of the primitive blocks world predicates such as *on*, *clear*, etc. In this paper, we aim to show that this convenience also makes it simpler to *learn* such concepts while simultaneously learning the policies based on them.

More precisely, through a number of experiments in the blocks world we obtain that

- the use of a different representation language has a significant impact in the effectiveness and generality of the approach; in particular, the concept language yields a better policy using less examples and *no background knowledge*
- the algorithm learns useful concepts and rules simultaneously, identifying domain concepts such as 'the-next-needed-block', 'a-well-placed-block', etc. from scratch
- the generalized policies using these learned concepts are closely related to the indexical-functional representations for action selection advocated by Agre and Chapman [1].

We also discuss current and inherent limitations of the approach and future work.

2 Policies

A problem in classical planning is given by a set of actions, an initial state, and a goal. These problems can be formulated as problems of search from the initial state to the set of goal states by applying actions that map one state into another. The result of this search is a path in state-space or a plan [20]. A related state-space formulation can be obtained by viewing planning from the perspective of dynamic programming [5]. In a dynamic programming formulation, the solutions of the planning problem is not an *action sequence* but a *policy* π that maps states into actions. The plan $a_0, a_1, \ldots, a_n$ that achieves the goal from an initial state s_0 can be obtained by setting a_0 to the action $\pi(s_0)$, a_1 to the action $\pi(s_1)$, and so on, where s_{i+1} is the result of applying action a_i in state s_i. Note that while a *plan* encodes the solution of a *single* planning instance, a *policy* encodes the solution of a *class* of instances; namely, the instances that differ from the original instance in the initial state.

Using the same ideas it is possible to represent solutions of larger classes of problems by defining suitable policies. For example, a policy defined over all states and all goals can encode the solutions of the class of problems that differ on both the initial state and the goal. Such policies map states and goals into actions.

In this paper, we are interested in finding the solution to a still larger class of problems. We are interested in learning how to solve *any* problem instance in a given domain. The type of policies needed to solve such large class of instances can be made precise by making explicit the description of planning problems. A planning problem is normally made up of two parts:

- a *domain description* in the form of some action *schemas*, and
- an *instance description* in the form of a set of object names, a state, and a goal

This distinction is explicit in the PDDL language, a recent standard for stating planning problems [18].

Given this distinction, a general policy for solving *all* problems in the domain must take an *instance description* as input, and map it into an action. We call such policies defined over all the domain instances, *generalized policies.*

The algorithms used for computing policies over the state-space cannot be generalized easily to the space of all instances. However, while no general method is known for computing such generalized policies, such policies can be defined for many domains. Such policies are not optimal but are general, can be applied efficiently (they involve no search), and often produce good results. For example, a generalized policy for the blocks world can be given as follows (see [14, 24] for better policies for this domain):

1. *put all blocks on the table, and then*
2. *move block x on block y when x should be on y, both blocks are clear, and y is well placed*

Here a block is 'well placed' when it is on the 'right' block (or table) and all blocks beneath it are well placed too. This strategy puts all blocks on the table and then moves them in order to their target positions. The strategy is completely *domain-dependent* (it only applies to the blocks-world), but is completely *instance-independent* (it applies to all the domain instances). Moreover, the length of the plans it produces is never more than twice the length of the optimal plans [14].

Similar policies can be defined for many planning domains; e.g., [11] formulates a general policy for logistics problems. An important question is whether such policies can be inferred automatically from domain descriptions. This question has been addressed recently by Khardon [16].

3 Learning Rule-based Policies

Khardon assumes a representation of generalized policies (that he calls *action strategies*) in terms of an ordered list of existentially quantified rules. For example, some of the rules look as:

$$Obj(x), Obj(y), clear(y),\ holding(x),\\ inplace(y), G(on(x,y)) \Rightarrow Stack(x,y)$$

The left-hand side of these rules expresses the conditions that are checked in the situation, and the right-hand side expresses the action to be taken when the conditions are true for certain bindings of the variables. A *situation* is a *problem instance*; i.e., a set of object names, a state, and a goal. The conditions that are preceded by the marker G are evaluated with respect to the goal, the object predicate Obj is evaluated with respect to the set of objects, and the other conditions are evaluated with respect to the state. The rule above says to stack an object x on top of an object y, if y is clear, x is being held, y is 'in place', and x is on top of y in the goal.

Khardon refers to an ordered list of such rules as a *production rule strategy* or PRS. The action determined by a PRS in a given situation is the action in the right-hand side of the first rule whose conditions are true in the situation. If several bindings of the variables match, the first binding (in some order) is selected.

The vocabulary for the rule conditions involves the *primitive predicates* that appear in the domain description (e.g., *on*, *clear*), *support predicates* defined by the user (e.g., *above*, *in_place*), the goal marker G, and the object predicate Obj.

Khardon shows how such policies can be learned from a training set given by a set of situation-action pairs obtained by solving a number of small problem instances. The learning algorithm is variation of Rivest's algorithm for learning decision lists [22]. In this algorithm, all *candidate rules* are first enumerated, and the PRS is obtained by selecting the rule that 'best' covers the instances in the training set, and iterating this procedure over the instances that are not covered.

```
For each candidate rule
  For each example in the training set
    If rule conditions satisfied by example
      Mark that rule covers example
      Test and Mark whether covers it
           correctly or incorrectly
Initialize Rule-List to empty
While training set not empty
  choose rule that best covers the examples
  add such rule to Rule-List
  remove examples covered by rule
```

Figure 1: Rivest's Learning Algorithm: the input is the training set and the set of candidate rules; the output is an ordered list of rules

The basic idea behind the algorithm is shown in Fig. 1. The algorithm is guaranteed to find an ordered list of rules compatible with the data if such list exists [22].

For the enumeration to work, the set of candidate rules need to be finite and not too large. This is achieved by setting limits to the number of conditions and variables allowed in each rule. The size of the resulting set of rules is the main source of complexity in the learning algorithm.

Khardon reports results in two domains: blocks-world and logistics. We focus on the blocks-world. For the experiments he uses rules with two conditions and three variables. The primitive predicates are $on(x,y)$, $clear(x)$, $on\_table(x)$, and $holding(x)$ and the actions are $pickup(x)$, $putdown(x)$, $unstack(x,y)$, and $stack(x,y)$. In addition to the primitive predicates, Khardon introduces four *support* predicates that encode *background knowledge* about the domain. These include the *above* predicate and the *in_place* predicate (that expresses that a block is not misplaced). Given a training set consisting of 4800 block-world instances involving 8 blocks and their solutions, the system infers a rule-based policy that solves 79% of the 8-blocks instances, 60% of the 15-block instances, and 48% of the 20-block instances. Interestingly, the planner used as a 'teacher' (Graphplan [6]) does not solve any of the large instances.

These results are encouraging and show that the performance of a planner can be boosted by using a learning component, and that generalized policies can be learned from data. While there have been a number of proposals for learning control knowledge to guide a search engine (e.g., [27]), few approaches have aimed at learning knowledge that completely eliminates the need to search.

The results obtained by Khardon, however, exhibit two weaknesses. First, the results are obtained using *support predicates* such as *above* and *in_place* that encode key features for solving the problem. This is domain-dependent knowledge that may be as difficult to provide as the domain-specific control knowledge that is sought. Second, the learned policies do not generalize so well: while solving 49% of the 20-block instance is better than what can be achieved by the domain-independent planner used a teacher, it is still far from what can be achieved with the simple policy for the blocks-world in which all blocks are put on the table and then stacked in order into their target positions.

In the rest of the paper we aim to show these weaknesses may be overcome by using an alternative language for representing policies.

4 Learning Concept-based Policies

Consider a control rule that says that 'if there is a clear block x that is misplaced, then move x to the table'. This rule has the form 'if an object is in a certain class, then do action a on the object' Our approach is based on the observation that rules of this form are very common in planning, and therefore, a language that makes the notion of *class* more central can provide a more compact description of generalized policies and a more convenient hypothesis space for learning them.

4.1 Concept Languages

The notion of classes of objects is central in the languages developed in AI known as *concept languages* or *description logics* [8, 9, 12]. These languages have the expressive power of subsets of standard first-order logic but with a *syntax* that is suited for representing and reasoning with classes of objects. From a logical point of view, one can think of concept languages as languages for defining complex predicates in terms of primitive ones. Predicates in concept languages are divided into two types: *unary predicates* or *concepts* that denote classes of objects, and *binary predicates* or *roles* that denote relations among objects. If we let C and C' stand for concepts, R and R' stand for roles, and C_p and R_p stand for *primitive* concepts and roles, the *complex* concepts can be defined by grammar rules of the form

$$C, C' \longrightarrow C_p \mid \top \mid \neg C \mid C \wedge C' \mid (\forall R.C) \mid R = R' \quad (1)$$

while complex roles can be defined as

$$R, R' \longrightarrow R_p \mid R_p^{-1} \mid R_p^* \mid R \circ R' \qquad (2)$$

Rule (1) says that concepts can be primitive concepts, the universal concept, the complement of a concept, the intersection of two concepts, the class of individuals in the domain whose R related individuals are all member of C, or the class of individuals in the domain whose whose R related individuals and whose R' related individuals are the same. Likewise, (2) says that roles can be primitive roles, the inverse or transitive closure of primitive roles, or the composition of two roles. The semantics of these constructions is formalized by defining the interpretations (extensions) C^I and R^I of complex concepts and roles in terms of the interpretation of its constituents and the domain of discourse Δ [12, 2]. As an illustration, the semantic clauses corresponding to some of the constructions above are

$$\begin{aligned} (\forall R.C)^I &= \{d_1 \in \Delta | \forall d_2 : \langle d_1, d_2\rangle \in R^I \rightarrow d_2 \in C^I\} \\ (R = R')^I &= \{d_1 \in \Delta | \forall d_2 \in \Delta \rightarrow (\langle d_1, d_2\rangle \in R^I \\ &\qquad \text{iff } \langle d_1, d_2\rangle \in R'^I)\} \\ (R_p^{-1})^I &= \{\langle d_2, d_1\rangle | \langle d_1, d_2\rangle \in R_p^I\} \\ (R_p^*)^I &= \{\langle d_1, d_3\rangle | \langle d_1, d_3\rangle \in R_p^I \text{ or} \\ &\qquad \langle d_1, d_2\rangle \in (R_p^*)^I \;\&\; \langle d_2, d_3\rangle \in R_p^I\} \end{aligned} \quad (3)$$

As an illustration, if on_s and on_g are primitive roles standing for the relation *on* in the current state s and in the goal G respectively, then $(on_s = on_g)$ is the concept that denotes the objects in the domain (blocks) that are on the same block in both s and G. Similarly, $(\forall on_g^*.\ on_s = on_g)$ is the concept that stands for the blocks that in the goal description are above blocks that are 'well placed'; i.e., all of which are on the same block in s and G. Note that equivalent descriptions in the standard syntax of predicate logic would be less compact.

4.2 Concepts for Planning

We assume that the planning domain is described in terms of a set of unary and binary predicates. For each predicate p, we create two predicates p_s and p_g: the first gets evaluated in the *state* of the given situation, the second gets evaluated in the *goal*.[1] From these *primitive* concepts and roles, the grammar above is used to generate *complex* concepts and roles. We limit the total set of concepts generated by imposing a limit on the *size* of the concepts, the size being understood as the number of connectives involved in the concept, or equivalently, the number of grammar rules needed to generate the concept. The notation $\mathcal{C}^i$ and $\mathcal{C}_i$ will be used to refer to the set of concepts of size i and size no greater than i respectively. For example, if the

[1] For predicates p that are not used in the goal description a single predicate p_s suffices.

concepts C_a and C_b have size 2 and 3, then the concept $C_a \wedge C_b$ will have size 6.

A *situation* given by a set of objects O, a state s , and a goal G provides an *interpretation* over the concept language. The domain of the interpretation Δ is given by the set of objects O, the interpretation p_s^I of the primitive predicates p_s is given by the state s, while the interpretation p_g^I of the primitive predicates p_g is given by the goal G. The interpretation C^I of all other concepts in $\mathcal{C}$ is given by the semantic rules (3). We say that a concept C is *satisfied* in a given situation I if $C^I \neq \emptyset$ and that o is an instance of C in I if $o \in C^I$.

For example, a goal description $on(a,b) \wedge on(b,c)$ defines the interpretation on_g^I of the primitive role on_g as the set containing the two pairs $\langle a,b\rangle$ and $\langle b,c\rangle$. The semantic clauses above then make the interpretation of the complex role on_g^* to be equal to the union of these two pairs and the pair $\langle a,c\rangle$.

4.3 Simple Concept-based Policies

In domains where all action predicates take a *single* argument, we define the *simple concept-based policies* as the ones expressed by an ordered list of concept-action pairs of the form:

$$C_1 : a_1, C_2 : a_2, C_3 : a_3, \ldots, C_m : a_m \tag{4}$$

where each C_i and a_i stand for a concept and an action predicate respectively. A policy of this form says to perform an action $a_i(o)$ in a situation I when o is an instance of the concept C_i and C_i is the first concept in the list that is satisfied in I. In other words, (4) says to do action a_1 on an object o_1 if there is one such object in C_1^I, else do action a_2 on o_2 if there is one such object in C_2^I, and so on.

Such policy is *non-deterministic* in the sense that the object o_i on which the action a_i is performed is chosen non-deterministically (randomly) from the class C_i^I. So during problem solving, the policy (4) can produce different actions when the same situation is encountered more than once.[2]

The policies defined by (4) are also *incomplete* in the sense that in a given situation it may be the case that none of the concepts C_i is satisfied and therefore no action is produced. We say in that case that the policy *fails.* This problem can be avoided by setting the last concept C_n in (4) to the universal concept $\top$ that is always satisfied. However, in the algorithm below for *learning* policies of the form (4) no special provision is made for precluding policies from failing in this way, and actually they very seldom do.

We call each of the concept-action pairs $C_i : a_i$, a *rule.* Note that if there are n_p action predicates and all concepts up to size n are considered, the number of candidate rules $C_i : a_i$ is $|\mathcal{C}_n| \cdot n_p$.

4.4 Non-simple Concept-based Policies

When actions have arity greater than 1, the simple concept-based policies defined by (4) do not apply. While we don't deal with such case in this paper, we will briefly discuss it in Section 7.

4.5 Learning Simple Concept-based Policies

We focus now on the algorithm for learning simple concept-based policies. The algorithm is a slight variation of the learning algorithm depicted in Fig. 1 from [22], where rules are of the form $C_i : a_i$, and examples in the training set are of the form $I : A$, where I is a situation (set of objects, state, and goal) and A is a *set* of actions. The distinctions from Khardon's approach are thus two: the rule language based on a concept language, and the use of *sets* of actions in the examples in the training set. The examples are obtained by running a planner on a set of small random instances and collecting *all* actions that are optimal in each of the situations encountered. The planner used is a modification of the HSP planner [7].

In order to apply the learning algorithm we have to specify when a rule $C_i : a_i$ *covers* an example $I : A$ and when the rule covers the example *correctly.* Thus we say that the rule $C_i : a_i$ covers the example $I : A$ when C_i^I is not empty, and that it covers the example $I : A$ correctly when for *every* object d in C_i^I, $a_i(d)$ is in A.[3] In other words, a rule covers the example when the rule applies to the example, and covers the example correctly when the set of actions it suggests are all compatible with the actions deemed appropriate in the example.

The rules $C_i : a_i$ that *best* cover a set of examples are obtained by considering several criteria: the number N_i of examples covered incorrectly by the rule, the number N_p of examples covered correctly, and the complexity of the rule (measured by the size of C_i).

[2] Such policies can be made *deterministic* by fixing an ordering among the objects in the domain and always selecting the first such object in C_i^I. See below.

[3] This definition is suitable for the non-deterministic interpretation of concept-based policies. For the deterministic interpretation where a fixed linear ordering among the objects is assumed a priori, it is more adequate to test only the *first* object d in C_i^I rather than *every* object.

These three measures are considered lexicographically: the second measure is maximized, while the first and third measures are minimized. Ties are broken randomly.

5 Experiments

5.1 Setting

The above ideas were applied to the problem of learning generalized policies in the blocks-world. The concept language is defined by the grammar rules (1) and (2) from the set of primitive concepts (unary predicates) $clear_s$, $clear_g$ and $holding_s$,[4] and the primitive roles (binary predicates) on_s and on_g. The actions considered are $pick(b_1)$, $put\_on(b_2)$ and $put\_on\_table(b_3)$, where b_1 refers to the block to be picked, b_2 refers to the block on which the block being held is to be placed, and b_3 denotes the block that is to be placed on the table.[5]

The universe of concepts was initially restricted to the class $\mathcal{C}_7$ of concepts with size up to 7. This set, however, contains several million concepts which turned out to be too large for our current Lisp prototype. So we decided to pick a subset of $\mathcal{C}_7$ by restricting the concepts in this subset to be *conjunctive* combinations of concepts C of size up to 3. In other words, the subset of concepts considered, that we call $\mathcal{C}_{3,7}$, stands for the concepts in $\mathcal{C}_7$ of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ such that each 'building block' C_i is a concept in $\mathcal{C}_3$. The choice of this set of concepts is largely arbitrary; yet much smaller sets such as $\mathcal{C}_5$ with less complex concepts produced poor results, while much larger sets such as $\mathcal{C}_8$ turn out to be intractable. Later on we will discuss how the selection of a suitable hypothesis space can be automated by an iterative search procedure in which the sets $\mathcal{C}_1$, $\mathcal{C}_2$, ... are considered in sequence.

The resulting number of concepts in the hypothesis space $\mathcal{C}_{3,7}$ is 72.228 after excluding redundant concepts such as $\neg\neg C$, $C \wedge C$, etc. We also removed concepts that were not sufficiently discriminating by generating a limited number of random situations (100) and excluding all concepts whose extensions over all these situations were equivalent to the universal or null concepts. We will say more about this below.

[4] The primitive concept $holding_g$ representing the class of objects being held in the goal was not used as the predicate *holding* does not appear in the goal descriptions of the instances considered.

[5] We chose this formulation because it makes all action predicates unary. In the standard formulations, some of these actions take two arguments, but while such arguments are needed in Strips, they are not needed in other action languages; e.g., [13].

Given the 3 unary action predicates in the domain and the 72,228 concepts, the total number of candidate rules $C\!:\!a$ considered was 216.684.

The training set was generated by solving 50 random instances with 5 blocks. For each situation arising in the solution of these problems we recorded the complete *set* of actions that were found to be optimal in each situation. This produced a set of 638 examples (situation-actions pairs).

5.2 Results

In our current Lisp prototype, computing the coverage of the 216.684 rules over the set of 638 examples takes several hours. This time can probably be reduced substantially but we haven't devoted much time to optimization. From the coverage information the computation of the list of rules $C_i\!:\!a_i$ representing the policy is very fast (Fig. 1).

Fig. 2 shows one of the typical policies obtained. As it can be seen, the policy is quite compact. We have written each rule $C_i\!:\!a_i$ as $a_i(C_i)$ to improve readability. Understanding the rules, however, requires some familiarity with concept languages. For example, the concept

$$(\forall\, on_g^{-1}.holding)$$

appearing in rule A1 expresses the class of blocks B such that the block being held goes on top of B in the goal. Rule A1 thus says to put the block currently being held on a block B such that B is on the same block in the state and the goal, the block being held goes on top of B in the goal, and B is currently clear. If there is no such block B, rule A1 does not apply, and the following rules are considered in order.

The second rule A2 says to put what's being held on the table. Rule A3 is interesting and says to pick up a block B if it's clear, and its target block C is clear and *well-placed.* This last condition is captured by the subconcept $(\forall\, on_g^*.(on_g = on_s))$. Also note that in the same rule the intersection

$$(\forall\, on_g^*.\ (on_g = on_s)) \wedge (\forall\, on_g.clear_s)$$

denotes the NEXT-NEEDED-BLOCK concept; namely, the blocks whose destination block is a clear and WELL-PLACED-BLOCK. The rest of the rules admit similar readings.[6]

[6] It's important to understand the rules in the context of the other rules. E.g., rule $A1$ appears as it may place a block on top of a 'bad' tower sometimes, however, this won't happen if the rules that select the blocks to be picked, pick up the 'correct' blocks.

A1: PUT_ON $((on_g = on_s) \wedge (\forall\ on_g^{-1}.holding) \wedge clear_s)$
A2: PUT_ON_TABLE $(holding)$
A3: PICK $((\forall\ on_g^*.(on_g = on_s)) \wedge (\forall\ on_g.clear_s) \wedge clear_s)$
A4: PICK $(\neg(on_g^* = on_s^*) \wedge (\forall on_s.(\forall on_g^{-1}.clear_s)) \wedge clear_s)$
A5: PICK $(\neg(on_g = on_s) \wedge (\forall\ on_g^*.(on_s^* = on_s)) \wedge clear_s)$
A6: PICK $(\neg(on_g = on_s) \wedge (\forall\ on_s.(\forall\ on_s^{-1}.clear_s)))$

Figure 2: Policy learned from initial set of examples

5 blocks	10 blocks	15 blocks	20 blocks
96.5%	87.6%	82.8%	72.2%

Table 1: Percentage of test problems solved by policy shown in Fig. 2 for different problem sizes

The policy in Fig. 2 was tested over a large set of randomly generated block-world problems of different size: 1000 problems of 5 and 10 blocks each, and 500 problems of 15 and 20 blocks each. These instances were generated using the program BWSTATES of Slaney and Thiébaux [25]. The percentage of problems solved by this policy is shown in Table 1.

The policy solves a problem when it leads to the goal within a maximum number of steps,[7] and otherwise fails (this includes situations where no rule applies). The coverage, as it can be seen from the table, is significantly better than the coverage obtained by the rule-based policies reported by Khardon, although it is significantly worst than the 100% coverage than can be achieved by the *simple policy* of putting all blocks on the table and then stacking the blocks in order. On the other hand, the *quality* of the solutions obtained by the learned policy is better on average than the quality of the simple policy, but of course, is not as good as the quality of the *optimal* solutions that can be found. These results are shown in Table 2 that displays the number of steps taken on average by each one of these three policies.[8] As it can be seen, the quality performance of the learned policy is closer to the performance of the optimal policy than to the performance of the simple policy.

[7] This maximum number of steps was defined as four times the number of blocks. This is the maximum number of actions (*pick*'s and *put*'s) that corresponds to the policy of moving all blocks to the table and then to their target positions.

[8] Optimal solutions to these problems were obtained using the optimal block-world solver BWOPT of Slaney and Thiébaux [25].

Policy	5 blks	10 blks	15 blks	20 blks
Learned	9.98	24.32	39.77	54.94
Simple	11.57	27.74	45.19	62.68
Optimal	9.94	23.48	37.5	51.60

Table 2: Average number of steps taken by the *learned* policy over the problems that are solved, in comparison with the *simple* policy of putting all blocks in the table first, and the *optimal* policy

B1: PUT_ON $((on_g^* = on_s^*) \wedge (\forall\ on_g^{-1}.holding) \wedge clear_s)$
B2: PUT_ON_TABLE $(holding)$
B3: PICK $((\forall\ on_g^*.(on_g = on_s)) \wedge (\forall\ on_g.clear_s) \wedge clear_s)$
B4: PICK $(((on_g \circ on_g) = on_g) \wedge (\forall\ on_s.(\forall\ on_s^{-1}.clear_s)))$
B5: PICK $(((on_g \circ on_g) = on_s) \wedge (\forall\ on_s.(\forall\ on_s^{-1}.clear_s)))$
B6: PICK $(\neg(on_g^{-1} = on_s) \wedge (\forall on_s.(\forall on_g.clear_s)) \wedge clear_s)$
B7: PICK $((\forall\ on_s.clear_g) \wedge clear_s)$
B8: PICK $(\neg(on_g^{-1} = on_s) \wedge (\forall on_s.(\forall on_g^{-1}.clear_s)) \wedge clear_s)$
B9: PICK $(\forall\ on_s.(\forall\ on_s^{-1}.clear_s))$

Figure 3: Policy learned after selective extension of training set

5.3 Incremental Refinement

The learned policy shown in Fig. 2 failed on 35 of the 1000 problems in the test set with 5 blocks. For these 35 problems, we computed the optimal solutions and detected the situations arising in these problems that were not covered correctly by the policy. This resulted in the identification of 32 *new* situations-actions pairs that were added to the training set. Since the coverage of the previous 638 examples was stored, only the coverage of the new 32 examples had to be computed for each of the 216.684 candidate rules. This refinement was much faster than the original computation, leading to a policy different than the one before, shown in Fig. 3.

The results of this new policy are shown in the second row of Table 3. The row displays the percentage of problems solved by the new policy on a new set of randomly generated problems. The new policy is better than the previous policy over the small problems but is worst over the larger problems. On the smallest 5-block problems it fails on 4 of the 1000 test problems. We performed the same refinement as before by computing the optimal plans for these 4 problems, identifying 4 new situation-actions pairs that were not accounted for correctly. These 4 examples were added to the training set and the learning algorithm was rerun resulting in a third policy whose performance is shown in the third row of Fig. 3. The new policy is better than the previous two policies and was computed very fast as only 4 new examples needed to be processed. The improvement obtained, however, is significant and has to do with the fact these few examples were not chosen randomly but were identified from the failures of the existing policy. This incremental learning process is akin to a form of *active* learning in which the training data is selected to boost performance while reducing the amount of data needed. In any case, as the second policy shows, the improvement is not monotonic.

Num. of exmpls	*Coverage of resulting policy*			
	5 blks	*10 blks*	*15 blks*	*20 blks*
638	96.5%	87.6%	82.8%	72.2%
670	99.6%	90.3%	82.0%	65.0%
674	99.7%	92.7%	84.2%	74.8%

Table 3: Percentage of problems solved by original learned policy and new policies obtained by incremental learning procedure described in text.

5.4 Deterministic vs. Non-Deterministic Policies

The results above improve the results obtained by Khardon [16] in the sense that the policies obtained produce better results while using a smaller set of examples and no background knowledge. We conjecture that this is because the concept language provides a more compact representation of generalized policies and therefore a more convenient hypothesis space for the learning algorithm to search. There is however an additional difference between Khardon's approach and ours. Khardon deals with *deterministic* policies where every situation produces a unique action and we deal with *non-deterministic* policies. We performed a quick test to evaluate how relevant this difference is by computing the coverage and quality of the learned policy shown in Fig. 2 when applied *deterministically.* For that, we fixed an ordering among the objects in the domain and always selected to act upon the first object in this ordering that satisfied the first concept in the policy list. The *coverage* of the resulting execution of the policy got indeed reduced while the average *quality* of the resulting plans improved slightly. The largest difference in coverage occurred in the largest problems. For the 20-block problems, the deterministic execution of the policy produced solutions for 59% of the problems, down from the 72.2% achieved by the non-deterministic execution. This is because the deterministic execution is more likely to get trapped into loops. By possibly choosing different actions when the same situation is encountered more than once, the non-deterministic execution breaks such loops. This also explains why the deterministic execution produces slightly shorter plans on average on the problems that it solves. The coverage of deterministic policies, however, can be improved by taking the ordering of the objects into account not only during the testing phase but also during the learning phase (when computing the coverage of each candidate rule). We are currently performing such experiments.

6 Related Work

This work presents an approach for learning generalized policies from data that is a variation of the approach taken by Khardon in [16]. Khardon represents generalized policies (action strategies) in a rule language and learns the policies in a supervised manner using a variation of Rivest's decision list learning algorithm [22] and additional background knowledge. We use the same learning algorithm but represent policies in terms of a concept language. The motivation for this is that planning involves the selection of the 'right' actions on the 'right' classes of objects, and concept languages provide a convenient, compact syntax for defining the useful classes of objects and learning them. Through a number of experiments in the blocksworld we have shown that the same learning algorithm produces better policies with less data and no background knowledge. We expect that similar results can be obtained in other richly structured domains even though there are a number of obstacles that must be overcome (see Sect. 7).

The approach described in this paper (as well as Khardon's) applies not only to classical, deterministic planning domains but also to *stochastic* planning domains where actions can be probabilistic.[9] In that case, the examples in the training set must be obtained by a dynamic programming procedure rather than by a classical planner. No other changes are required and the same procedures can yield suitable policies over such domains. In this sense, it's worth comparing this approach with *reinforcement learning* methods that are also concerned with learning policies for classical and stochastic planning problems [26]. One difference is that our approach is *supervised* as it relies on the existence of set of small *solved* instances while reinforcement learning (RL) is *unsupervised* (it doesn't need

[9] A (fully observable) planning problem with probabilistic action is a Markov Decision Process (MDP) whose solution can be cast as a policy that maps states into actions [5, 23].

a teacher). A second difference is that in reinforcement learning the representation of the policy takes the form of a *value function.* While value functions can generalize across different states and even across different goal descriptions, they cannot cope easily with changes in the size of the state space as when we move from a problem with 5 blocks to a problem with 20 blocks. Thus, value functions do not appear suitable for learning *generalized* policies.

The concept-based representation of generalized policies has the flavor of the functional-indexical representations of policies advocated by Agree and Chapman [1]. However, in Agree's and Chapman's approach, useful concepts like the NEXT-NEEDED-BLOCK are defined by the programmer and their denotation is assumed to be provided by sensors. In the approach described here, the concepts are learned from the examples and their denotation is given by their logical structure. The two ideas, however, are related; a concept-based policy can be executed faster when the denotation of the top concepts needs not be inferred from the denotation of the basic predicates but is provided directly by sensors. Thus the top concepts appearing in the policy express the aspects of the world that are actually worth sensing. Moreover, in the concept language as in functional-indexical representations, objects are not referred by their particular names (e.g., block A, B, etc) but by the role they play in going from the current state to the goal.

The use of learning algorithm on top of rich, logical representations, relates this work to the work in *Inductive Logic Programming.* For example, FOIL is a system that learns first-order rules from data in a supervised manner [21]. Like Rivest's learning algorithm, FOIL selects the 'best' rules one by one, eliminating the examples that are covered, and iterating this procedure on the examples that are left. The main difference is in the way the 'best' rules are selected. In Rivest's algorithm, this is done by an *exhaustive* evaluation of *all* rules. This provides a completeness guarantee (if there are decision lists consistent with the data, the algorithm will find one such list) but does not scale up to very large rule sets. The greedy approach, on the other hand, does not provide guarantees but can be used over much larger rule sets. We have followed the first approach.

Finally, it's worth mentioning the work on computing generalized policies by evolutionary methods. For example, both Koza in [17] and Baum in [4], develop evolutionary algorithms for obtaining generalized policies for versions of the blocks world. In Koza's approach, the policies are represented by Lisp programs, while in Baum's approach they are represented by a collection of rules in a syntax adapted to the application. While the results obtained in both cases are good, a comparison is not easy as both approaches appear to rely on domain-specific background knowledge in the form of useful domain features ('the-next-needed-block' [17]; 'top-of-the-stack' [4]) or domain-specific state representations. Like RL methods, these evolutionary methods are non-supervised, but unlike RL methods, they can compute generalized policies over all problem instances in the domain.

7 Discussion

Building on the work of Khardon [16], we have presented a representation for generalized policies based on a concept language and have shown through a number of experiments in the blocks-world that the new representation appears to produce better policies with less training data and no background knowledge.

The challenge is to show that these ideas can be applied successfully to other structured planning domains. The main obstacle is the combinatorics of the learning algorithm that requires computing the denotation of each one of the concepts in $\mathcal{C}_n$ in each one of the situations in the training set. The number $|\mathcal{C}_n|$ of concepts of size up to n grows exponentially with n. Currently, we consider very small values of n and exclude a number of redundant concepts such as $\neg\neg C$, $C \wedge C$, etc. However this pruning is not enough in general. We could use a more complete subsumption test over concepts but it's not clear whether such tests are cost-effective. An alternative approach that we plan to explore in the future is to generate the sets $\mathcal{C}_n$ incrementally, for $n = 0, 1, \ldots$, pruning all concepts that 'appear' equivalent to simpler concepts. This can be done quickly over a set of random examples: two concepts 'appear' equivalent when they have the same extensions over each one of the examples. While such pruning is not sound, it may allow us to deal with richer domains with more predicates with a small penalty in the quality of the policies obtained. An alternative that seems less appealing is to move away from Rivest's algorithm, replacing the exhaustive search and evaluation of *all* possible rules by an incomplete search.

Another limitation of the approach presented in this paper is the arbitrary choice of the set $\mathcal{C}_n$ of concepts considered. However, the incremental pruning approach can provide a solution to this problem: rather than chosing the value of n a priori, we can do an iterative search procedure, evaluating the policies resulting from the set of concepts $\mathcal{C}_1$, $\mathcal{C}_2$, $\ldots$ in sequence, until a good policy is found.

A more serious limitation is the restriction to *unary* actions. One option there is to break actions of higher arity into several actions of lower arity and suitable fluents and preconditions. This transformation can be done automatically in a domain-dependent manner but it may not be effective in general. A second option is to extend the rules $C_i : a_i$ in the simple policies (4) by rules of the form $C_i^1, C_i^2, \ldots, C_i^m : a_i$, where m is the arity of the action predicate a_i. The tuple of arguments upon which the action a_i is done is then selected from the corresponding tuple of concepts. This idea is simple but does not exploit the fact that the action arguments are usually functionally related (e.g., move a clear block to *its* target location). A more promising approach is to allow the concept C_i^k that stands for the k-th argument of the action to refer to the previous arguments C_i^j, $j < k$. Thus, if the number m is a reference to the m-th action argument, an action like 'moving a clear block to its target position' would be expressible as $move(clear_s, (\forall on_g^{-1}.\ 1))$.

A final issue that is worth mentioning is that in some domains more powerful concept languages may be needed; e.g., languages involving number restrictions or min-max operators. We hope to address this and the other limitations discussed in future work.

Acknowledgments

Part of this work was done while H. Geffner was visiting IRIT, Toulouse, and Linköping University. He thanks J. Lang, D. Dubois, H. Prade, and H. Farreny in Toulouse, and E. Sandewall and P. Doherty in Linköping for their hospitality and for making these visits possible. We also thank J. Slaney and S. Thiébaux for making the programs BWSTATES and BWOPT available. The work has been partially supported by grant S1-96001365 from Conicit, Venezuela and the Wallenberg Foundation, Sweden.

References

[1] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings AAAI-87*, 1987.

[2] F. Baader and U. Sattler. Number restrictions on complex roles in description logics. In *Proceedings KR'96*. Morgan Kaufmann, 1996.

[3] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning, 1998. Submitted.

[4] Eric Baum. Toward a model of mind as a laissez-faire economy of idiots. In *Proceedings Int. Conf. on Machine Learning*, 1996.

[5] D. Bertsekas. *Dynamic Programming and Optimal Control, Vols 1 and 2.* Athena Scientific, 1995.

[6] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, Montreal, Canada, 1995.

[7] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of ECP-99*. Springer, 1999.

[8] R. Brachman and H. Levesque. The tractability of subsumption in frame based description languages. In *Proceedings AAAI-84*, 1984.

[9] R. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation systems. *Cognitive Science*, 9(2), 1985.

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

[11] F. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings KR'91*. Morgan Kaufmann, 1991.

[12] H. Geffner. Functional strips: a more general language for planning and problem solving. Presented at the Logic-based AI Workshop, Washington D.C., June 1999.

[13] N. Gupta and D. Nau. Complexity results for blocks-world planning. In *Proceedings AAAI-91*, pages 629–633, 1991.

[14] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, pages 1194–1201, 1996.

[15] R. Khardon. Learning action strategies for planning domains. Technical Report TR-09-97, Harvard, 1997. To appear in AI Journal.

[16] J. Koza. *Genetic Programming I.* MIT Press, 1992.

[17] D. McDermott. PDDL – the planning domain definition language. Available at `http://ftp.cs.yale.edu/pub/mcdermott`, 1998.

[18] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings IJCAI-99*, 1999.

[19] B. Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3), 1988.

[20] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, 1980.

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

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

[23] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1994.

[24] J. Slaney and S. Thiébaux. Linear time near-optimal planning in the blocks world. In *Proceedings of AAAI-96*, pages 1208–1214, 1996.

[25] J. Slaney and S. Thiébaux. Software for the block-world: BWSTATES and BWOPT. At `http://arp.anu.edu.au/ jks/bw.html`, 1999.

[26] R. Sutton and A. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[27] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Find, and J. Blythe. Integrating learning and planning: The prodigy architecture. *J. of Experimental and Theoretical AI*, 1994.