Learning Generalized Plans Using Abstract Counting

Siddharth Srivastava and **Neil Immerman** and **Shlomo Zilberstein**
Department of Computer Science
University of Massachusetts,
Amherst, MA 01003
{siddharth, immerman, shlomo}@cs.umass.edu

Abstract

Given the complexity of planning, it is often beneficial to create plans that work for a wide class of problems. This facilitates reuse of existing plans for different instances drawn from the same problem or from an infinite family of similar problems. We define a class of such planning problems called generalized planning problems and present a novel approach for transforming classical plans into generalized plans. These algorithm-like plans include loops and work for problem instances having varying numbers of objects that must be manipulated to reach the goal. Our approach takes as input a classical plan for a certain problem instance. It outputs a generalized plan along with a classification of the problem instances where it is guaranteed to work. We illustrate the utility of our approach through results of a working implementation on various practical examples.

Introduction

In many real word problems, it is beneficial to find solutions that are parametric in the numbers of objects involved, rather than searching for independent solutions for different instances. For example, in the blocks world, a single plan or algorithm that un-stacks a tower of blocks of any height is better than having a planner that can solve the problem for each height independently. This issue generalizes all the way to programming situations that typically require the ability to deal with unknown or variable quantities in the input (e.g. reversing a linked list of unknown length; sorting a linked list with data). We present a unified framework which can be used to express such general problems, to generalize sample classical plans for solving them, and also to find generalized plans directly through search in an abstract state space. We focus on the learning capabilities in this paper.

Typically, it is difficult to identify when an example plan (or a plan being generated) includes loop un-rollings because it is difficult to characterize what makes successive states similar. This makes it difficult to conclude that a condition, or a loop invariant holds after every few action steps. We use a state-abstraction technique from static analysis of programs (Sagiv, Reps, & Wilhelm 2002) to make the identification of loop un-rollings very easy. Using this technique we present a novel algorithm for generalizing a given classical plan which works for some instance of the problem. We also characterize a class of problem domains where we can compute the set of problem instances for which our learned generalized plan will be guaranteed to work. In our experience this usually covers large portions of the problem instances, leaving out only a few of the infinitely many. Our expression for the general problem also makes it easy to find the right kind of examples needed for generalization.

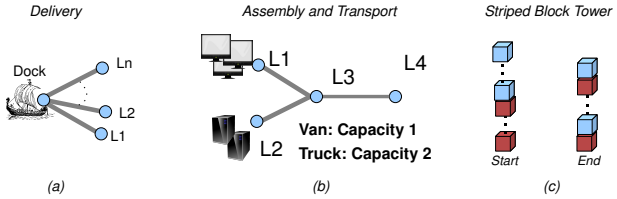

Figure 1: Example problem domains

Our contributions include: a system for representing generalized planning problems; a unified framework for solving these problems through search and by generalizing example plans; an approach for learning generalized plans, with a prototype implementation and results; we also provide an analysis of our techniques and a class of domains where we can provide preconditions for our generalized plans.

Generalized Planning Problems

We define a generalized planning problem as follows. Given a goal condition and a class of initial states, the objective is to find a plan that works for a subset of the solvable instances. Typically the input class has states with varying numbers of elements, which necessitates that the generalized plan be parametric in these numbers. More precisely, if states are described as structures of First-Order logic, then the generalized problem consists of the vocabulary, the action operators and first-order formulas representing the initial and goal conditions. An instance of this problem would be an extension of the vocabulary with any number of constant symbols for each class or unary predicate, and a model satisfying the initial state formula.

For example, consider a problem from the transport domain (Fig. 1(b)). We have two source locations L1 and L2, which have a variable number of monitors and servers respectively. There are two types of transport, a van at L1 and a truck at L2 with capacities 1 and 2 respectively. For sim-

plicity, in this example we have exactly one vehicle of each kind. The generalized planning problem is to deliver *all* – regardless of the actual numbers – items to L4, but only in pairs with one item of each kind. This abstracts an assembly and transport domain, where the two objects have to be combined before delivery. We consider generalizations of the plan in which the van moves a monitor from L1 to L3 and returns to L1; the truck then takes a server from L2 to L4, while picking up the monitor at L3 on the way. Such a plan for 10 pairs of items consists of 110 actions. Together with the range of instantiated actions, results discussed later show that the depth of search involved makes a direct plan-search impractical as the number of objects (or locations, see Delivery example discussed later) increases. Given an example plan for six pairs of objects, our algorithm discovers the loop and returns a generalized plan that works for infinitely many instances (its precondition is that we have at least 3 items of each kind, with matching counts). This example can be extended to include variable truck capacities, delivery locations, a more complicated graph structure, etc. Although such problems are common enough in real life, we believe they haven't received sufficient study or even characterization. There is also no "database" of such problems, familiar to the community and readily available for comparison.

Prior Approaches Attempts at producing plans for more than a single problem instance started with (Fikes, Hart, & Nilsson 1972). Their framework parametrized and indexed subsequences of existing plans for use as macro operations or alternatives to failed actions. However, this approach turned out to be quite limited and prone to over-generalization. More recent approaches addressing such problems have included using prior domain knowledge of the looping constructs that may be encountered (Shavlik 1990), searching for patterns in iteratively generated plans (Levesque 2005), and using plan annotations for recognizing repeated applications followed by a compilation the results into domain-specific planners (Winner & Veloso 2003). Levesque addresses problems that involve a unique planning parameter. Our approach allows a more general notion of counters. Also, unlike his approach, we find provably correct plans. However, our approach does not fully accommodate numeric fluents, which are handled in his approach. The approach presented by Winner & Veloso is limited to plan learning and is directed towards solving all problems in a domain. It also does not guarantee correctness. Fern, Yoon, & Givan (2006) developed an approach for generating policies for solving all problems of a domain. Their approach requires extensive initial training spanning several hours. It also used a relatively restrictive goal description language.

In contrast to these approaches, our work is focused on solving well-defined classes of problem instances. As we demonstrate later in the paper, the generalized problems we work with can capture several interesting practical problem-classes where classical planners do not scale well. Our approach can also be applied directly to some program synthesis problems which up to now have needed expensive deduction based approaches. Most of the approaches discussed above do not provide a clear characterization of what constitutes a general problem, how to express it, and more importantly, when the learned plans or policies work. Lack of problem characterization makes it difficult to select the right training examples for these approaches. Absence of classifications of solved instances makes generalizations difficult to use as re-planning could be required at any time during plan execution.

Overview Our approach for generalizing example plans is as follows. We first use an abstraction technique based on unary predicates to collapse similar concrete states together. This allows us to construct a finite abstract state space for efficiently modeling states from infinitely many instances of the general problem. This process, together with our state representation and action mechanisms is described in the next section. Next, we trace out an example plan's effects in the abstract state space. Due to abstraction, similar states become apparent and we are able to detect loops. This is described in the section on "Algorithm for Generalizing an Example Plan". Finally, if the given domain is what we call an extended-LL domain (all examples in this paper qualify), we compute the preconditions for our generalized plan. This is presented together with our analysis and categorization of extended-LL domains in the "Finding Preconditions" section. We present the results of our prototype implementation on various problems in the last section.

Framework

We assume that actions are deterministic and that their results are observable.

Representation

States of a domain are represented by two-valued structures in first-order logic with transitive closure (FO[TC]), consisting of a universe of constant elements or objects and definitions for all the predicates in a domain-specific vocabulary. We use $[\![\varphi]\!]^S$ to denote the truth value of a closed formula φ in the structure S. State transitions are carried out using action operators described as a set of formulas in FO[TC], defining new values of every predicate in terms of the old ones. We represent abstract states using structures in three-valued logic ("abstract structures"). While the terms "structure" and "state" are interchangeable in our setting, we will use the former when dealing with a logic-based mechanism.

Example 1 An example structure, S, for the transport problem discussed above can be described as: the universe, $|S| = \{i_1, i_2, t_1, t_2\}$, $monitor^S = \{i_1\}$, $server^S = \{i_2\}$, $Van^S = \{t_1\}$, $Truck^S = \{t_2\}$, $atL_1^S = \{i_1, t_1\}$, $atL_2^S = \{i_2, t_2\}$, $inVan^S = inTruck^S = \emptyset$. Unlike the delivery problem discussed later, a non-relational encoding was possible here because the number of locations is fixed.

The action operator for an action (e.g., $a(\bar{x})$) consists of a set of preconditions and a set of formulas defining the new value p' of each predicate p. Action arguments are selected in pre-action steps. For instance, for the $Load$ action, the pre-action steps set up predicates identifying the vehicle to be loaded and the object to load. These predicates are used to bind the variables obj_1 and obj_2 to the

vehicle to be loaded and the object to load respectively. Using this notation, the preconditions for $Load$ can be $\{\vee_{i\geq 1}(atL_i(\text{obj}_1)\wedge atL_i(\text{obj}_2))\}$. Vehicle capacities would also be modeled here. We use the action operator without operands (e.g., a) to represent an instantiated action.

Let Δ_i^+ (Δ_i^-) be formulas representing the conditions under which the predicate $p_i(\bar{x})$ will be changed to true (false) by a certain action. The formula for p_i', the new value of p_i, is written in terms of the old values of all the relations:

$$p_i'(\bar{x}) = (\neg p_i(\bar{x}) \wedge \Delta_i^+) \quad \vee \quad (p_i(\bar{x}) \wedge \neg\Delta_i^-) \quad (1)$$

The RHS of this equation consists of two conjunctions, the first of which holds for arguments on which p_i is changed to true by the action; the second clause holds for arguments on which p_i was already true, and remains so after the action. These update formulas resemble successor state axioms in situation calculus. However, we use query evaluation on possibly abstract structures rather than theorem proving to derive the effect of an action.

We use a to denote a complete action operator including the precondition test and the action update (more components will be added to deal with abstract structures); τ_a denotes just the predicate update part of a. This separation is helpful while dealing with abstract structures. We use $\tau_a(\Gamma)$ to denote application of τ_a to a set of states, Γ.

Example 2 The Assembly and Transport domain has the following actions: $\cup_{i=1\ldots4}\{moveToL_i\} \cup \{Load, Unload\}$. Update formulas for the $Load$ action are as follows:

$$inVan(x) = inVan(x) \vee (\text{obj}_2 = x \wedge Van(\text{obj}_1))$$
$$inTruck(x) = inTruck(x) \vee (\text{obj}_2 = x \wedge Truck(\text{obj}_1))$$

The goal condition is represented as a formula in FO[TC]. For example, $\forall x(monitor(x) \vee server(x)) \implies atL_4(x)$. We define domain-schemas as follows:

Definition 1 A *domain-schema* is a tuple $\mathscr{D} = (\mathcal{V}, \mathcal{A}, \varphi_g)$ where $\mathcal{V}$ is a vocabulary, $\mathcal{A}$ a set of action operators, and φ_g, a formula in FO[TC] representing the goal condition.

Given a domain-schema, some special unary predicates are classified as *abstraction predicates*. The special status of these predicates arises from the fact that they are preserved in the abstraction. We define the *role* an element plays as the set of abstraction predicates it satisfies:

Definition 2 A *role* is a conjunction of literals consisting of every abstraction predicate or its negation.

Example 3 In all our examples, we let the set of abstraction predicates be all the unary predicates used for the problem representation. In the transport domain, the role representing objects of type $monitor$ at L_1 that are not in any vehicle is: $monitor(x) \wedge atL_1(x) \wedge \wedge_{i>1}(\neg atL_i(x)) \wedge \neg inVan(x) \wedge \neg inTruck(x)$. For simplicity, we will omit the negative literals and the free variable when expressing a role. The role above then becomes $monitor \wedge atL_1$.

Abstraction Using 3-valued Logic

We perform state abstraction using *canonical abstraction*, a technique originally developed for TVLA, which is a tool for static analysis of programs (Sagiv, Reps, & Wilhelm 2002). This technique abstracts a structure by merging all objects of a role into a *summary object* of that role. The resulting abstract structure is used to represent concrete structures having a positive number of objects for every summary object's role. Since the number of roles for a domain is finite, the total number of abstract structures for a domain-schema is finite; we can tune the choice of abstraction predicates so that the resulting abstract structures effectively model some interesting general planning problems and yet the size and number of abstract structures remains manageable. For generalizing plans however, we do not need a full state-space search and all the unary predicates can be treated as abstraction predicates.

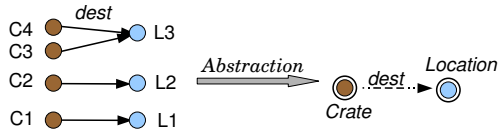

Figure 2: Abstraction in the delivery domain

The imprecision that must result when objects are merged together is modeled using three-value logic. In a three-valued structure the possible truth values are $0, \frac{1}{2}, 1$, where $\frac{1}{2}$ means "don't know". If we order these values as $0 < \frac{1}{2} < 1$, then conjunction evaluates to minimum, and disjunction evaluates to maximum.

Example 4 Consider a delivery problem where some crates (satisfying the predicate $Crate$) need to be delivered to various retail locations (satisfying the predicate $Location$) from a dock. The functional relation $dest(crate_i, loc_j)$ represents the destination of each object. Fig. 2 shows the result of abstraction on this relation when only $Crate$ and $Location$ are used as abstraction predicates. The dotted edge represents the $\frac{1}{2}$ truth value, and the encapsulated nodes denote summary objects.

In order to define canonical abstraction more precisely, we first define *embeddings* (Sagiv, Reps, & Wilhelm 2002). Define the *information order* on the set of truth values as $0 \prec \frac{1}{2}, 1 \prec \frac{1}{2}$, so lower values are more precise. Intuitively, S_1 is embeddable in S_2 if S_2 is a correct but perhaps less precise representation of S_1. In the embedding, several elements of S_1 may be mapped to a single summary element in S_2.

Definition 3 Let S_1 and S_2 be two structures and $f : |S_1| \rightarrow |S_2|$ be a surjective function. f is an *embedding* from S_1 to S_2 ($S_1 \sqsubseteq^f S_2$) iff for all relation symbols p of arity k and elements, $u_1, \ldots, u_k \in |S_1|$, $[\![p(u_1, \ldots, u_k)]\!]^{S_1} \preceq [\![p(f(u_1), \ldots, f(u_k))]\!]^{S_2}$.

The universe of the canonical abstraction, S', of structure S, is the set of nonempty roles of S. The truth values in canonical abstractions are as precise as possible: if all embedded elements have the same truth value then this truth value is preserved, otherwise we must use $\frac{1}{2}$.

In order to merge all elements that have the same role, we use the subscript $\{p \in A | [\![p(x)]\!]^{S,u/x} = 1\}, \{p \in A | [\![p(x)]\!]^{S,u/x} = 0\}$ in the following definition to denote elements in the abstracted domain.

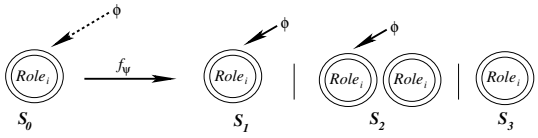

Figure 3: Effect of focus with respect to ϕ.

Definition 4 The embedding of S into its *canonical abstraction* wrt the set A of abstraction predicates is the map:

$$c(u) = e_{\{p \in A | [\![p(x)]\!]^{S,u/x}=1\},\{p \in A | [\![p(x)]\!]^{S,u/x}=0\}}$$

Further, for any relation r, we have $[\![r(e_1,\ldots,e_k)]\!]^{S'} = l.u.b_{\preceq}\{[\![r(u_1,\ldots,u_k)]\!]^S | c(u_1)=e_1,\ldots,c(u_k)=e_k\}$.

The set of concrete structures that can be embedded in an abstract structure S is called the *concretization* of S: $\gamma(S) = \{S' | \exists f : S' \sqsubseteq^f S\}$.

Focus With such an abstraction, the update formulas for actions might evaluate to $\frac{1}{2}$. We therefore need an effective method for applying action operators while not losing too much precision. This is handled in TVLA using the *focus* operation. The focus operation on a three-valued structure S with respect to a formula φ produces a set of structures which have definite truth values for every possible instantiation of variables in φ, while collectively representing the same set of concrete structures, $\gamma(S)$. A focus operation with a formula with one free variable is illustrated in Fig. 3: if $\phi()$ evaluates to $\frac{1}{2}$ on a summary element, e, then either all of e satisfies ϕ, or part of it does and part of it doesn't, or none of it does. This process could produce structures that are inherently infeasible. Such structures are either refined or discarded during TVLA's *coerce* operation using a set of restricted first-order formulas called *integrity constraints*. In Fig. 3 for instance, if integrity constraints restricted ϕ to be unique and satisfiable, then structure S_3 in Fig. 3 would be discarded and the summary elements for which $\phi()$ holds in S_1 and S_2 would be replaced by singletons. These two structures represent the cases where we have either exactly one, or more than one object with e's role. Using focus in this manner allows us to model the "drawing-out" of individuals from their summary elements prior to action update.

The focus operation wrt a set of formulas works by successive focusing wrt each formula in turn. The result of the focus operation on S wrt a set of formulas Φ is written $f_\Phi(S)$. We use ψ_a to denote the set of focus formulas for action a.

Choosing Action Arguments

Since abstract structures collapse objects into roles, in the pre-action steps we employ the focus operation to draw out individual objects to be used as action arguments. For example, if we had to choose a monitor to load for the $Load$ action, the pre-action step would focus on an auxiliary unary predicate $choose\_\mathrm{obj}_2()$ that is constrained to be single-valued and is initialized to $\frac{1}{2}$ for objects satisfying $monitor(x)$. In the implementation, we initialize $choose\_\mathrm{obj}_2()$ to $\frac{1}{2}$ for all objects satisfying $monitor(x) \vee server(x)$, in order to allow all possible choices.

Action Application

Even with the action arguments drawn out from their roles through the pre-action steps, predicate update formulas for actions can evaluate to $\frac{1}{2}$ due to imprecision in relations. For our purposes, the most important updates are for (unary) abstraction predicates. Recall that the predicate update formulas for an action operator take the form shown in equation 1. For unary predicate updates, the expressions for Δ_i^+ and Δ_i^- are *monadic* (i.e. have only one free variable apart from the action arguments which are bound by the pre-action steps). Therefore, in order to obtain definite truth values for these updates, we focus the given abstract structure using $\Delta_i^\pm$ as the focus formulas. The resulting focused structures are tested against the preconditions, and action updates are applied to those that qualify. The resulting structures are then canonically abstracted (this is called "blur" in TVLA), yielding the abstract result structures.

This completes the framework required to model a planning problem in the abstract domain.

Transitions

To summarize the discussion above, once the action arguments have been chosen, there are three steps involved in action application: action specific focus, action update, and blur. The action operator $a(\bar{x})$ collectively represents all of these steps. We define transition relations $\xrightarrow{a}$ to capture the effect of action operators:

Definition 5 *(Transition Relation)* $S_1 \xrightarrow{a} S_2$ iff S_1 and S_2 are three-valued structures and there exists a focused structure $S_1^1 \in f_{\psi_a}(S_1)$ s.t. $S_2 = blur(\tau_a(S_1^1))$.

Sometimes we will need to study the exact path S_1 took in getting to S_2. For this, the transition $S_1 \xrightarrow{a} S_2$ can be decomposed into a set of transition sequences $\{(S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2) | S_1^i \in f_{\psi_a}(S_1) \wedge S_2^i = \tau_a(S_1^i) \wedge S_2 = blur(S_2^i)\}$.

Algorithm for Generalizing an Example Plan

In this section we present our approach for computing a generalized plan from a plan that works for a single problem instance. The idea behind this technique is that if a given concrete plan contains sufficient un-rollings of some loops, then we can automatically identify them by tracing the example plan in the abstract state space and looking for identical abstract state and action sequences. We can then enhance this plan using the identified loops and use the techniques discussed in the next section to find the set of problem instances for which this new generalized plan will work. The procedure is shown in Algorithm 1.

Suppose we are given a concrete example plan $\pi = (a_1, a_2, \ldots, a_n)$ for a concrete state $S_0^\#$. Let $S_i^\# = a_i(S_{i-1}^\#), i > 0$. Let S_0 be any structure which makes the resulting domain extended-LL, and for which we need a generalized plan; the canonical abstraction of $S_0^\#$ forms a natural choice. To obtain the generalized plan, we first convert π into a sequence of action operators by replacing each action with an operator that uses as its argument(s) any element

Algorithm 1: GeneralizeExample

Input: $\pi = (a_1, \ldots, a_n)$: plan for $S_0^\#$; $S_i^\# = a_i(S_{i-1}^\#)$
Output: Generalized plan Π

1 $S_0 \leftarrow canAbs(S_0^\#)$; $\Pi \leftarrow \pi$; $C_\Pi \leftarrow \top$
2 $\{S_0; a_1, \ldots, S_{n-1}; a_n, S_n\} \leftarrow$ $Trace(\{S_0^\#; a_1, \ldots, S_{n-1}^\#; a_n, S_n^\#\})$
3 $\Pi \leftarrow$ formLoops$(S_0; a_1 \ldots, S_{n-1}; a_n, S_n)$
4 **if** $\exists C \in \mathcal{C}_I(\mathcal{R}) : S_n|_C \models \varphi_g$ **then**
5 $\quad C_\Pi \leftarrow$ findPrecon(S_0, Π, φ_g)
6 return Π, C_Π

having the role (or even just the type) of the concrete action's argument(s). We successively apply the operators from this sequence to S_0, at every step keeping only the abstract structure S_i that embeds the concrete structure $S_i^\#$ for that step. This process, which we call *tracing*, is implemented in the $Trace$ subroutine.

The $formLoops$ subroutine converts a linear path of structures and actions into a path with simple (i.e, non-nested) loops. The restriction to simple loops is imposed so that we can efficiently find plan-preconditions. One way of implementing this routine is by making a single pass over the input sequence of abstract-state and action pairs, and adding back edges whenever $\langle S_j, a\rangle$ is found such that $\langle S_i, a\rangle = \langle S_j, a\rangle (i < j)$, and $\langle S_i, a\rangle$ is not part of, or behind a loop. Structures and actions following S_j are merged with those following S_i if they are identical; otherwise, the loop is exited via the last action edge. This method produces one of the possibly many simple-loop paths from π; we could also produce all such paths. $formLoops$ thus gives us a generalization Π of π. Steps 4 and 5 of the algorithm deal with finding preconditions: if a subset of the final abstract structure satisfies the goal, then we use the $findPrecons$ subroutine to obtain the restrictions on S_0 for which Π works.

Finding Preconditions

For both searching for plans from scratch and learning from examples, we need to be able to find the concrete states that a given sequence of actions, possibly with loops, can take to the goal. In order to accomplish this, we need a way of representing regions of abstract states that are guaranteed to take a particular branch of an action's focus operation. We also need to be able to pull these subsets backwards through action edges in the given path all the way up to the initial abstract state – thus identifying its "solved" concrete members. We represent regions of an abstract structure by annotating it with a set of conditions from a chosen constraint language.

Definition 6 *(Annotated Structures)* Let $\mathcal{C}$ be a language for expressing constraints on three-valued structures. A *$\mathcal{C}-$annotated structure* $S|_C$ is the refinement of S consisting of structures in $\gamma(S)$ that satisfy the condition $C \in \mathcal{C}$. Or, $\gamma(S|_C) = \{s \in \gamma(S) \mid s \models C\}$.

We extend the notation defined above to sets of structures, so that if Γ is a set of structures then by $\Gamma|_C$ we mean the structures in Γ that satisfy C. Thus, $\gamma(S|_C) = \gamma(S)|_C$.

Given an action, the annotated pre-image of a resulting structure $S_2|_{C_2}$ constitutes the preconditions for reaching $S_2|_{C_2}$ through that action. There are two aspects of finding these pre-images: first, we need to classify structures in S_1 that will take the appropriate focus branch; second, we need to rollback the condition C_2 itself. If these operations are possible, we say the domain is *amenable to back propagation*:

Definition 7 *(Annotated Domains)* An *annotated domain-schema* is a pair $\langle \mathscr{D}, \mathcal{C}\rangle$ where $\mathscr{D}$ is a domain-schema and $\mathcal{C}$ is a constraint language. An annotated domain-schema is *amenable to back-propagation* if for every transition $S_1 \xrightarrow{f_{\psi a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$ and a target condition $C_2 \in \mathcal{C}$ there is an annotation $C_1^i \in \mathcal{C}$ which selects the pre-image of $S_2|_{C_2}$ along this transition. In other words, there should be an annotation C_1^i such that $\tau_a(\gamma(S_1)|_{C_1^i}) = \tau_a(\gamma(S_1^i))|_{C_2}$.

In terms of this definition, since $\tau_a(\gamma(S_1^i))$ is the subset of $\gamma(S_2^i)$ that has pre-images in S_1^i under τ_a, $S_1|_{C_1^i}$ is the pre-image of $S_2|_{C_2}$ under a particular focused branch (the one using S_1^i) of action a. The disjunction of C_1^i over all branches taking S_1 into S_2 therefore gives us a more general annotation which is not restricted to a particular branch of the action update.

By induction, we can compose back-propagation along linear sequences of actions in domains that are amenable to back-propagation. This makes it straightforward to compute preconditions of linear sequences of actions in any such domain (detailed proofs of this fact and the other results in this section can be found at (Srivastava, Immerman, & Zilberstein 2007)). However, we need to simplify $\mathcal{C}$ in order to find preconditions of plans with loops.

Inequality-Annotated domain-schemas Let us denote by $\#_R(S)$ the number of elements of role R in structure S. In this paper we use $\mathcal{C}_I(\mathcal{R})$, the language of constraints expressed as sets of linear inequalities using $\#_{R_i}(S)$, for annotations.

Quality of Abstraction In order for us to be able to classify the branches caused by focus operations, we need to impose some quality-restrictions on the abstraction. Our main requirement is that the changes in abstraction predicates should be characterized by roles: given a structure, an action should be able to change a certain abstraction predicate only for objects with a certain role. We formalize this property as follows: a formula $\varphi(x)$ is said to be *role-specific in* S iff only objects of a certain role can satisfy φ in S.

Definition 8 *(Extended-LL domains)* An *Extended-LL domain with start structure* S_{start} is a domain-schema such that Δ_i^+ and Δ_i^- are role-specific, exclusive when not equivalent, and uniquely satisfiable in every structure reachable from S_{start}. More formally, if $S_{start} \rightarrow^* S$ then $\forall i, j, \forall e, e' \in \{+, -\}$ we have Δ_i^e role-specific and either $\Delta_i^e \equiv \Delta_j^{e'}$ or $\Delta_i^e \implies \neg\Delta_j^{e'}$ in S.

Handling Paths with Loops In extended-LL domains we can also effectively propagate annotations back through paths consisting of simple (non-nested) loops:

Proposition 1 (Back-propagation through loops) *Suppose*

$S_0 \xrightarrow{\tau_1} S_1 \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_{n-1}} S_{n-1} \xrightarrow{\tau_0} S_0$ *is a loop in an extended-LL domain with a start structure* S_{start}*. Let the structures before entering the loop and after exit be* S *and* S_f*. We can then compute an annotation* $C(l)$ *on* S *which selects the structures that will be in* $S_f|_{C_f}$ *after* l *iterations of the loop on* S*, plus the simple path from* S *to* S_f*.*

Theorem 1 *Extended-LL Domains are amenable to back-propagation.*

Detailed proofs of these results can be found at (Srivastava, Immerman, & Zilberstein 2007).

Methods described by Srivastava, Immerman, & Zilberstein can be used to find plan preconditions in extended-LL domains. Intuitively, these domain-schemas are those where:

1. The information captured by roles is sufficient to determine whether or not an object of any role will change roles due to an action; and
2. The number of objects being acquired or relinquished by any role is fixed (constant) for each action.

Implementation and Results

We implemented a prototype for our approach using TVLA as an engine for computing results of action sequences. The implementation proceeds in four phases. In the first phase, the example plan is executed on the concrete initial state that it came with. This gives us a sequence of concrete state and action pairs. In the second phase, the example plan is executed on an abstract start state, with the real operands replaced by unary predicates representing their classes (e.g. Red or $item$). In the third phase, for every action we select the unique abstract result structure that embeds the corresponding result structure from the concrete run (tracing). Finally, loops are identified as described above.

Our planner is named *Aranda* [1]. We used Python for the implementation. `Pyparsing` was used to parse the transition graphs produced by TVLA for phase 3. `Pyparsing` is known to be a slow parser, and this operation is the most expensive in our implementation. `pyparsing` takes about 5 minutes to parse a file containing $2,500$ structures. This is extraneous to our algorithm: an exponentially faster system can be implemented by following the tracing procedure described in the section on our Algorithm. This requires a better interface with TVLA.

Results

We ran Aranda on some practical problems derived from classical planning benchmarks. We summarize the problems and the results below.

Delivery We have some objects at a Dock, which have to be transported to various retail locations. Every object has a specific destination, determined by the $dest$ relation. The Dock is connected to all the retail locations (Fig. 1(a)). There is only one truck, initially at the garage. The generalized planning problem is to find a plan to deliver all the objects, irrespective of the number of objects or retail stores. We used `findDest` action in this domain to focus on $dest(obj, x)$ thereby "sensing" the destination of an object, and setting the predicate $itemDest$ to it.

[1] after an Australian tribe that had names for numbers only up to two.

```
01. setDest(dock)
02. go()
03. choose(item)
04. findDest()
05. Load()
06. setDest(itemDest)
07. go()
08. unload()
09. setDest(dock)
10. go()
11. choose(item)
12. findDest()
13. load()
14. setDest(itemDest)
15. go()
16. Repeat:
17.   unload()
18.   setDest(dock)
19.   go()
20.   choose(item)
21.   if (#item = 1)
21a.    exit loop
22.   findDest()
23.   load()
24.   setDest(itemDest)
25.   go()
26. findDest()
27. ...
```

Figure 4: Generalized Plan for Delivery

```
//Actions up to moving Truck to L4
01. Repeat:
02.   Unload()
03.   mvToL3()
04.   mvToL2()
05.   chooseVehicle(Van)
06.   chooseItem(monitor; atL1)
07.   if #(monitor; atL1) = 1
07a.    exit loop
08.   Load()
09.   mvToL3()
10.   Unload()
11.   mvToL1()
12.   chooseVehicle(Truck)
13.   chooseItem(server; atL2)
14.   Load()
15.   mvToL3()
16.   chooseItem(monitor; atL3)
17.   Load()
18.   mvToL4()
19. Load() ...
```

Figure 5: Main loop for Assembly and Transport

The input example plan delivered five objects to two different locations. Our algorithm found the generalized plan shown in Fig. 4. Actions 26 to 39 execute the last iteration of the loop. In all of our examples, it is possible to merge the loop iterations before and after the identified loop by checking that their result structures are embeddable in the structures inside the loop. Since we are in an extended-LL domain, we can use the methods described by (Srivastava, Immerman, & Zilberstein 2007), to compute the preconditions for this plan as $\#(item) \geq 3$. Essentially, this is because the loop exit occurs at statement 20 due to the focus operation, when the number of items to choose from is exactly 1. In fact, here and in all the following examples the preconditions also show how many loop unrollings there will be in a plan execution (e.g. $\#(item) = l + 3$, where $l \geq 0$ is the number of loop iterations).

Assembly and Transport This problem was described in the second section. We represented this domain without using any binary relations. The example plan for this problem was described in the second section. Fig. 5 shows the main loop discovered by our algorithm. Action operands in the loop are just the roles of chosen objects in the corresponding structures. Preconditions can be computed as $\#(monitor \wedge atL_1) = \#(server \wedge atL_2) = l + 3$.

Striped Block Tower Given a tower of red and blue blocks with red blocks at the bottom and blue blocks on top, the goal is to find a plan that can construct a tower of alternating red and blue blocks, with a red "base" block at the bottom and a blue block on top. We used transitive closure to express stacked towers and the goal condition. The input example plan worked for six pairs of blocks, by first unstacking the whole tower, and then placing blocks of alternating colors back above the base block. Our algorithm discovered three loops: unstack red, unstack blue, stack blue and red (Fig. 6). The preconditions indicate that for the plan to work, we must have $\#(Red) = \#(Blue) = l + 4$. The representation for this problem can also be used to model, and

```
1. choose(topmost)
2. mvToTable()
3. choose(topmost)
4. mvToTable()
5. Repeat:
6.    choose(topmost)
7.    mvToTable()
8.    if (#Blue = 1)
8a.      exit loop
9. choose(topmost)
10. mvToTable()
11. choose(topmost)
12. mvToTable()
13. choose(topmost)
14. mvToTable()
15. Repeat:
16.    choose(topmost)
17.    mvToTable()
18.    if (#Red = 1)
18a.      exit loop
19. choose(topmost)
20. mvToTable()
21.. // axns for mv Blue, an
unrolling of the next loop//
26. Repeat:
27.    choose(Red;onTable)
28.    if (#Red;onTable = 1)
28a.      exit loop
29.    choose2(Blue;topmost)
30.    move()
31.    choose(Blue;onTable)
32.    choose2(Red;topmost)
33.    move()
34. choose(Blue;onTable)...
```

Figure 6: Generalized Plan for Striped Block Tower

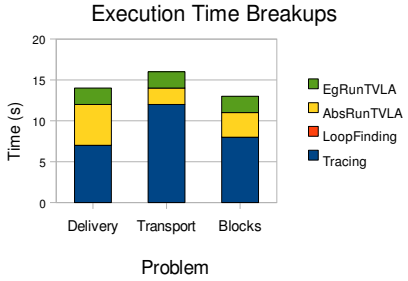

Figure 7: Break-ups of Aranda's plan generalization times

solve program synthesis problems like reversing and traversing link lists. This gives us a method for solving such problems without the computational overheads of automated deduction.

A summary of the timing results can be seen in Fig. 7. The parsing time for phase 3 in Aranda was between 4 and 6 minutes for these examples and is not shown in the plots. Loop finding times were less than a second.

Comparison We compared Aranda's plan-generalization times with planning times of winners of the last International Planning Competition (IPC-5 2006): SGPlan5 (a non-optimal planner) and SATPLAN06 (an optimal planner). We had to explicitly ground our problem encodings for these planners, which sometimes made the problem easier. For example, the goal position of each block in the Striped Tower problem became explicitly known. Although SGPlan5 allowed quantified goals, they make the planner very unscalable – with a quantified goal formula stating the alternating color condition, it ran for 1500 seconds without solving a Striped Tower problem instance with 10 pairs.

We present a summary of the results in Fig. 8. The results clearly illustrate the utility of learning generalized plans. We used vanilla versions of the planners for this comparison. The sensing aspect of the Delivery problem could not be accurately translated without quantified goals, and was thus not included in the comparison. All but the larger runs for SGPlan were carried out on a 1.6GHz Pentium Dual Core machine with 1.5GB RAM. SGPlan with 30 or more pairs of objects had to be run on a 2GHz machine with 2GB RAM.

Conclusion and Future Work

In this paper we introduced a new category of planning problems and presented a novel approach for learning generalized plans for solving them. Our framework allows us to obtain the class of problem instances that these generalized plans solve. The applicability and utility of our approach

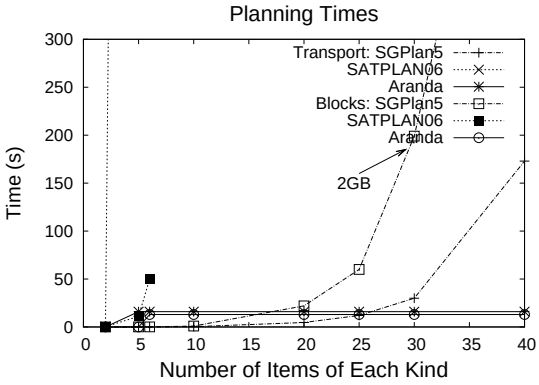

Figure 8: Comparison of plan generalization and planning times

was illustrated through results on several practical problems from classical planning-benchmark domains. Other than the natural extension of our work beyond extended-LL domains, open directions include learning plans with branches inside loops. Such generalized plans can be easily expressed and searched for in our framework, but the current methods cannot learn them from example plans. Developing standardized evaluation metrics for the quality of such plans is also a very interesting problem and would be invaluable to the community for comparing approaches that provide plans for solving multiple problem instances. Although we can detect nested loops, succinctly expressing preconditions for such loops is left for future work.

Acknowledgements

Support for this work was provided in part by the National Science Foundation under grants CCF-0541018 and IIS-0535061.

References

Fern, A.; Yoon, S.; and Givan, R. 2006. Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes. *Journal of Artificial Intelligence Research* 25:85–118.

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and Executing Generalized Robot Plans. Technical report, AI Center, SRI International.

IPC-5. 2006. Fifth International Planning Competition. `http://zeus.ing.unibs.it/ipc-5/`.

Levesque, H. J. 2005. Planning with Loops. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Sagiv, M.; Reps, T.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems* 24(3):217–298.

Shavlik, J. W. 1990. Acquiring Recursive and Iterative Concepts with Explanation-Based Learning. *Machine Learning* 5:39–70.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2007. Using Abstraction for Generalized Planning. Technical report, 07-41, Dept. of Computer Science, Univ. of Massachusetts, Amherst.

Winner, E., and Veloso, M. 2003. DISTILL: learning domain-specific planners by example. In *Proc. of the International Conference on Machine Learning*.