Using Abstraction for Generalized Planning

Siddharth Srivastava
Thesis Advisors: Neil Immerman and Shlomo Zilberstein
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01002

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 of the same problem or even for other problems that are somehow similar. My dissertation work focuses on a novel approach for finding and learning such plans using state representation and abstraction techniques originally developed for static analysis of programs. Our algorithm for finding generalized plans takes as input a finite 3-valued first-order structure representing a set of initial states and a goal test. The output is a set of *generalized plans* and conditions describing the problem instances for which they work. These plans–which could include loops and are thus close to algorithms–work for a large class of problem scenarios having varying numbers of objects that must be manipulated to reach the goal. The algorithm for learning generalized plans takes a plan and a similar 3-valued structure as its input and produces a similar generalized plan along with a description of the cases for which it works.

Introduction

Consider the problem of unstacking a tower of hundred blocks. A conventional state-of-the-art planner with a reasonable heuristic has no trouble finding its solution. Now consider the task of unstacking a tower of blocks of height between zero and hundred. If deduction or theorem proving is not used, a state search based planner would typically have to be run with the equivalent of a hundred different state spaces to solve this problem. Such limitations challenge the scalability of current planners, even on trivial tasks.

The goal of my dissertation work is the development of theory and implementation for planners that can compute general solutions for input problem classes. More specifically, given an abstract problem description consisting of a set of initial states and a goal condition, our planners will find a general (possibly conditional) plan for reaching a goal state from any of the initial states. The initial states could come from different state spaces corresponding to different instances of a general problem, e.g. constructing a pattern of blocks, or reversing a linked list. In the rest of this document, we refer to this problem as generalized planning.

So far, we have formalized a framework for generalized planning using state-space search and have developed precise algorithms for search and for learning from examples in this framework. We have also worked out an interesting class of problem domains where our methods for finding general plans succeed, and have rigorous proofs for this classification. We have a partial implementation of the algorithm for finding generalized plans and are currently working on a complete implementation.

Related Work

Approaches for generalized planning typically attempt to *learn* a generalization by parametrizing parts of available plans and assimilating or merging them. Our approach is novel in its ability to perform a direct state-space search for generalized plans.

Interest in generalizing plans can be traced back to Fikes *et al.* [1972], who presented a framework for parametrizing and indexing 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. Recent approaches include DISTILL[Winner & Veloso, 2003], which works by attaching newly found plans into existing templates, with support for very limited kinds of loops (having just one action). This approach however does not provide any guarantees of correctness. Levesque [2005] presents an interesting method for creating general plans from scratch. He introduces a planning parameter which counts the objects due to which generalization is required (e.g., number of blocks in the tower). Our approach is more general in its use of multiple such counters which can be created and destroyed during search. Also, unlike his approach, we find provably correct plans. However, our approach cannot currently handle numeric fluents, which are accommodated in Levesque's approach.

Setting

We assume that actions are deterministic and that their results are observable. States of a domain are represented by two-valued first-order logic structures consisting of constant elements or objects, and definitions for all the predicates in a domain-specific vocabulary. State transitions are carried out using an action's *transformer*, which is a set of first-order formulas describing new values of every predicate in terms of the old values. We use three-valued logic structures, which we call abstract structures, to represent sets of two-valued structures succinctly.

The input to our planning algorithm is an abstract structure representing a set of initial states of interest, and a goal condition in first-order logic. The output is a conditional plan and the set of abstract structures for which it is guaranteed to work. The input for our plan learning algorithm is an example concrete plan generated by a typical classical planner and a goal condition. The output is a generalized plan with loops and the set of abstract structures for which it is guaranteed to work. Currently, we only produce plans with conditions on the number of objects with certain domain-specific key properties which are parameters to the abstraction mechanism. This restriction comes from the pre-condition evaluation phase, discussed later.

In this paper we assume that a generalized plan is constructed entirely before execution, but it is possible to model a sensing agent where irrelevant plan branches are discarded as the action results are observed.

General Idea

The general, high level idea of our approach is as follows:

Abstraction Use an abstraction scheme to represent multiple state spaces corresponding to different domain instances or different "similar" problems in one *finite* state space.

Action Mechanism Define a *sound* action mechanism. That is, the result of an action transformer on an abstract state S includes the results of that action on each of its member states.

A more restricted action mechanism could also be used: one that under-approximates the results on application upon an abstract state. Such a formulation may not find plans when they actually exist; we focus on sound mechanisms in this paper.

Search Use the action mechanism in the abstract state space to search for paths with simple, non-nested loops to states satisfying the goal condition. When a path is found, find its pre-conditions. Continue this search until (a) all paths are exhausted, or (b) entire range of input states is covered.

Loops are allowed because in an abstract state space they can be progressive. Our pre-condition evaluation step evaluates if the loops in a path make progress towards the goal.

Methodology

Representation

We represent states of a domain using logical structures. A structure, S, of vocabulary $\mathcal{V}$, consists of a universe $|S| = \mathcal{U}$ along with a relation R^S over $\mathcal{U}$ for every relation symbol R in $\mathcal{V}$ and an element $c^S \in \mathcal{U}$ for every constant symbol in $\mathcal{V}$.

Example 1 A typical blocks world vocabulary would consist of a binary relation on; this can be used to define other relations like $onTable$ and $topmost$ using first-order formulas. For clarity in presentation however, we will treat all of these relations as separate and equally fundamental. An example structure, S, in this vocabulary can be described as: $|S| = \{b_1, b_2, b_3\}$, $onTable^S = \{b_3\}$, $topmost^S = \{b_1\}$, $on^S = \{(b_1, b_2), (b_2, b_3)\}$.

Abstraction We use *canonical abstraction* from the TVLA system [Sagiv *et al.*, 2002], which has been very successful in verifying properties of heap and pointer manipulating programs. Canonical abstraction creates an abstract state by constructing equivalence classes of elements

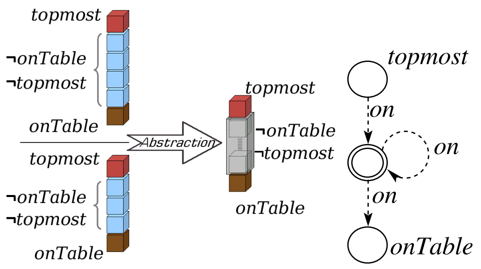

Figure 1: Canonical abstraction in blocks world. Abstracted objects are encapsulated. The abstraction predicates are $topmost$ and $onTable$; diagram on the right shows the state in TVLA notation.

of a structure on the basis of values of certain special unary predicates called *abstraction predicates*. Abstraction predicates are specific to a domain and are important parameters for generating useful abstractions. For example, in the blocks world $\{topmost, onTable\}$ can be abstraction predicates. All blocks in the middle of a stack would satisfy $\neg onTable \wedge \neg topmost$, and thus fall into a single equivalence class (Fig. 1). Abstract structures represent such classes as elements satisfying the $summary$ predicate. These elements are drawn with double circles in the figures.

The resulting imprecision in truth values of other predicates is handled using 3-valued logic. For instance, Fig.1 represents all block towers of height at least three. on holds between the topmost block, e_1, and some but not all of the blocks of the summary element, e_2. The truth value of $on(e_1, e_2)$ therefore needs to be imprecise. Three valued logic is used for this in TVLA, and $on(e_1, e_2)$ is given the truth value $\frac{1}{2}$, drawn in TVLA as a dotted arc. First order formulas called *integrity constraints* enable us to unpack such structures into their concrete components when needed. For a theoretical description, see [Srivastava *et al.*, 2007; Sagiv *et al.*, 2002].

Canonical abstraction always produces finite state spaces. We can tune the choice of abstraction predicates so that abstract structures effectively model some interesting general planning problems and yet the size and number of abstract structures remains manageable.

Action Mechanism The mechanism for computing action effects via transformers carries over to abstract structures, and its soundness follows from the embedding theorem [Sagiv *et al.*, 2002].

Example

Figure 2 shows a structure in TVLA notation - relations with truth values $\frac{1}{2}$ are shown using dotted edges and summary elements are shown using double circles.This structure represents all problem instances with a single stack of unknown and unbounded numbers of red and blue blocks (at least two of each). The blue blocks are above the red blocks. With this as our initial structure, our goal is to reach a state with all blocks stacked up in an alternating red and blue pattern, with a red block at the bottom and a blue block at the top.

When executed on the structure in Fig.2, our algorithm finds a plan that works for input towers with any number red and blue blocks, as long as they are equal. The equality pre-condition is also computed automatically. The most interesting branch of this general plan is shown in Fig.3, which

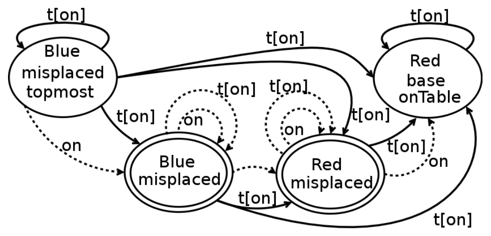

Figure 2: Initial structure for striped blocks world.

```
1 Move mispld. blk to table
2 while $\#$(blue blocks in middle) $> 1$
    Move mispld. blk to table
3 Move mispld. blk to table
4 Move mispld. blk to table
  //All Blue blocks on table
5 Move mispld. blk to table
6 while $\#$(red blocks in middle) $> 1$
    Move mispld. blk to table
7 Move mispld. blk to table
8 Move mispld. blk to table
  //All blocks on table
9 Move blue blk to stack
10 Move red blk to stack
11 Move blue blk to stack
12 Move red blk to stack
13 Move blue blk to stack
14 while $\#$(blue on table $> 1$)
     Move red blk to stack
     Move blue blk to stack
15 Move red blk to stack
16 Move blue blk to stack
```

Figure 3: Generalized plan with three loops found through a search in the abstract state space.

works for input towers having 4 red and blue blocks. Cases with fewer blocks are captured by other paths to the goal, found prior to this path. The branch in Fig.3 consists of three loops: the first loop moves all blue blocks to the table; the second, the red blocks, and finally, the last loop moves blue and red blocks alternately back on to the stack. Our algorithm also finds an additional pre-condition for this branch: the number of red and blue blocks should be at least 4.

Learning From Examples

This framework can also be used to learn a generalized plan from examples. Instead of using only the patterns in action traces, abstraction allows us to find similar structures also. We can construct generalized plans by running an example plan on a chosen initial abstract structure and including the loops found in the abstract state and action trace. Using the techniques for evaluating pre-conditions, we can then classify the initial structures for which this general plan works; continuing in this way and using more examples enables us to span the space of initial structures contained in the chosen initial abstract structure.

For example, the plan in Fig.3 can be recovered from a concrete plan for an instance of the problem described above with five each of red and blue blocks. This concrete plan could have been computed by any classical planner. This process is described in detail in [Srivastava *et al.*, 2007].

Theoretical Results

So far, we have identified an interesting class of domains which we call *extended-LL* domains, where our algorithm can find generalized plans with any number of simple loops. These domains include problems from programming languages such as linked list reversal, block world problems, rocket-domain problems, problems for building products from ingredients etc. Rigorous proofs for our results are available in the full version [Srivastava *et al.*, 2007].

Contributions so far

1. A new framework for generalized planning. Our framework bridges the areas of planning and program synthesis by using state-space search for algorithm-like plans.
2. Algorithms for finding pre-conditions of generalized plans in our framework.
3. An Algorithm for computing provably correct generalized plans. This can be applied to program synthesis tasks.
4. Formal description of a set of domains where our techniques for finding generalized plans and their pre-conditions work.
5. An algorithm for learning provably correct generalized plans from examples.
6. Implementation the search module for abstract state spaces (we are working on a full implementation).

Future Work and Conclusions

Our framework addresses two major difficulties faced by current approaches in this direction: (a) guaranteeing correctness and (b) reliance upon automated deduction in the absence of examples. By computing solutions for classes of problems, we foray into the area of algorithm synthesis.

This direction of work presents several interesting ideas and open questions for further research. Searching in abstract state spaces presents new challenges and requires a different class of heuristics. The equivalent of admissible heuristics in this setting is would be very useful. The two aspects discussed in this paper, viz. searching from scratch and generalizing from examples are closely related but independently worthy of research. Theoretical guarantees on when our techniques can learn generalized plans, and of their quality would also be interesting. Extending our methods beyond extended-LL domains is also an obvious direction for further research. For my thesis, I intend to focus on extending our techniques to a broader class of domains, developing a full implementation of our methods and conducting theoretical analyses of the scope and properties of our approaches for finding and learning generalized plans.

A more detailed description of our work has been accepted in the ICAPS workshop on AI Planning and Learning (2007). A full version containing all the proofs and algorithms is available at [Srivastava *et al.*, 2007].

References

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

Levesque, H. J. 2005. Planning with Loops. *In Proc. of IJCAI.*

Sagiv, M.; Reps, T.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*.

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

Winner, E., and Veloso, M. 2003. Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*.