Termination and Correctness Analysis of Cyclic Control

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

The utility of including cyclic flows of control in plans has been long recognized by the planning community. Loops in a plan help increase both its applicability and the compactness of representation. However, progress in finding such plans has been limited largely due to lack of methods for reasoning about the correctness and safety properties of loops of actions. We present an overview of recent results for determining the class of problems that a plan with loops can solve. These methods can be used to direct the construction of a rich new form of generalized plans that solve a desired class of problems.

1. Introduction

Automated planning is one of oldest problems in AI (Fikes and Nilsson 1971). In spite of significant advances since its early origins, fundamental limitations on the scalability of planning have led to a growing interest in finding generally applicable, or "generalized" plans. In contrast to traditional representations, generalized plans typically include cyclic flows of control and can be efficiently instantiated to solve infinitely many problem instances with unknown, and unbounded quantities of objects. As a simple example, a generalized plan with a loop over an *unload()* action can unload a truck that has an unspecified number of objects. This is a goal that no linear plan can achieve. Using such constructs in plans can significantly increase the scope and applicability of automated planning.

We present fundamental techniques for determining the conditions under which such plans terminate and achieve some desired goals. Although determining these conditions is incomputable in general, our results identify the factors that make this problem difficult and characterize very useful sub-classes that are solvable. These techniques can be used in the search for generalized plans as well as for computing the set of problem instances that are solved by a given generalized plan.

A planning problem is typically defined as the problem of constructing a sequence of actions which, when executed starting at a given initial state, will lead to a given goal state. In a partially observable formulation, the problem becomes one of constructing a *contingent* plan, or a tree with actions labeling nodes and observations labeling edges (outgoing edges from a node are labeled with possible observations, and each edge leads to the action that must be executed if the corresponding observation occurs).

Typically, the worst-case complexity of finding a plan for a given problem grows as the space of possible plans grows. The size of this plan-space is exponential in the maximum size of plans that have to be considered, which in turn is proportional to the size of the state space of a given problem. Therefore, one potential approach for increasing the scalability of planning is to make the size of a plan required to solve a problem independent of the size of its state space. The central motivation behind using cyclic flows of control in generalized plans is to achieve this independence. Generalized plans are typically represented in the form of "controllers" or finite-state machines with observations as inputs and actions as outputs. By employing loops of actions, generalized plans represent patterns of activity (in a manner similar to regular expressions) of unbounded length, rather than the (possibly contingent) sequences of actions of fixed lengths represented by acyclic plan representations.

Generalized Plans: Advantages and Challenges

Recent work in planning shows almost ubiquitous advantages of using generalized plans. Such plans have been used and computed in various contexts: to provide handwritten *domain control knowledge* to aid automated planners (Baier et al. 2007); to compute succinct policy representations for solving situations with partial observability (Bonet et al. 2009); in tasks like programming by demonstration, where an algorithm or a program is constructed by generalizing example executions (Srivastava et al. 2011a); and for automated service composition (Bertoli et al. 2010), in addition to the explicit purpose of solving broad classes of problems (Levesque 2005; Winner and Veloso 2007; Srivastava et al. 2011b).

However, all these benefits of generalized plans come at a significant cost: generalized plans can in general represent algorithms. Therefore finding the class of problem instances that a generalized plan can solve, or even just determining if a generalized plan will terminate is *undecidable* in general. Another consequence of this limitation is that it is impossible in general to compute a generalized plan for solving a *given* class of problems of interest. Therefore, although any plan with a cyclic flow of control that terminates will solve some class of problem instances (a pattern of activity naturally corresponds to the pattern of problem instances that

it solves), it is impossible in general to find such plans for solving an arbitrary class of problems.

This is a severe limitation in finding generalized plans. Not surprisingly, very few approaches assert correctness of the generalized plans they produce, or even compute generalized plans for solving a given class of problems (Hu and Levesque 2010; Bertoli et al. 2010; Srivastava et al. 2011b; 2011a). In this paper we present a general method for determining the conditions under which plans with loops will terminate at a particular state (Srivastava et al. 2010).

We initially assume that planning actions come from a simple, but powerful, class of action operators, which can only increment or decrement a finite set of registers by unit amounts. Although this class of actions appears to be too primitive for planning domains, plans with loops for many interesting planning problems can be automatically translated into plans with such actions (Srivastava et al. 2011b).

The class of actions considered in this work is captured by *abacus programs*—an abstract computational model as powerful as Turing machines. The halting problem for abacus programs is thus undecidable. In other words, finding closed-form applicability conditions, or preconditions for plans with loops of just the primitive abacus actions is undecidable. Despite this negative result, we show that closed-form preconditions *can* be found very efficiently for certain structured classes of abacus programs, and demonstrate that such structures are sufficient to solve interesting planning problems. In addition to determining the scope of applicability of existing plans, these methods can also be used during plan computation for identifying useful loops of actions.

2. Abacus Programs

Abacus programs are finite automata whose states are labeled with actions that increment or decrement a fixed set of registers (Lambek 1961). Formally,

Definition 1. (Abacus Programs) *An abacus program* $\langle \mathcal{R}, \mathcal{S}, s_0, s_h, \ell \rangle$ *consists of a finite set of registers* $\mathcal{R}$*, a finite set of states* $\mathcal{S}$ *with special initial and halting states* $s_0, s_h \in \mathcal{S}$ *and a labeling function* $\ell : \mathcal{S} \setminus \{s_h\} \mapsto$ Act*. The set of actions,* Act*, consists of actions of the form:*

- $Inc(r, s)$*: increment* $r \in \mathcal{R}$*; goto* $s \in \mathcal{S}$*, and*
- $Dec(r, s_1, s_2)$*: if* $r = 0$ *goto* $s_1 \in \mathcal{S}$ *else decrement* r *and goto* $s_2 \in \mathcal{S}$

We represent abacus programs as bipartite graphs with edges from nodes representing states to nodes representing actions and vice-versa. State-nodes have at most one outgoing edge and action-nodes have at most two outgoing edges; the two edges out of a decrement action are labeled $= 0$ and > 0 respectively (see Fig. 1). A more succinct representation that does not use state-nodes is also possible, but we use state-nodes to improve clarity and maintain a correspondence with planning scenarios.

Given an initial valuation of its registers, the execution of an abacus program starts at s_0. At every step, an action is executed, the corresponding register is updated, and a new node is reached. An abacus program *terminates* iff its execution reaches the halt node. The set of final register values in this case is called the *output* of the abacus program.

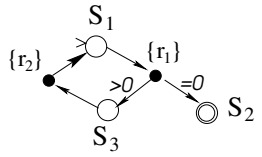

Figure 1: A simple abacus machine for the program: `while` $(r_1 > 0)$ { $r_1 - -; r_2 + +$}

Abacus programs are equivalent to Minsky Machines (Minsky 1967), which are as powerful as Turing machines and thus have an undecidable halting problem:

Fact 1. *The problem of determining the set of initial register values for which an abacus program will reach the halt node is undecidable.*

Nevertheless, we identified a general class of abacus programs for which the halting problem *is* decidable.

As discussed in the introduction, our approach for determining the utility and applicability conditions of loops of planning actions is to view them as abacus programs with the same loop structure. Maintaining the loop structure while doing this translation is important as it will allow us to selectively construct plans with the structures that we know can be handled. However, the abacus program framework is restrictive from this point of view: it does not include non-deterministic actions. In planning on the other hand, non-deterministic sensing actions are common and we need a way to translate them into the abacus framework effectively, without changing the loop structure. For this purpose, we extend the abacus program framework with the following non-deterministic action, in the representation of Def. 1:

Definition 2. (Non-deterministic Abacus Programs) *Non-deterministic abacus programs are abacus programs whose set of actions,* Act *includes, in addition to the* Inc *and* Dec *actions, non-deterministic actions of the form:*

- NSet(r, s_1, s_2)*: set* r *to* 0 *and goto* $s_1 \in \mathcal{S}$ ***or*** *set* r *to* 1 *and goto* $s_2 \in \mathcal{S}$*.*

where $\mathcal{S}$ *is the set of states of the abacus program and* r *is a register that is not used by deterministic actions.*

A non-deterministic action thus has two outgoing edges in the graph representation, corresponding to the two possible values it can assign to a register. Either branch may be taken during execution. Although the original formulation of abacus programs is sufficient to capture any computation, these actions will allow us to conveniently treat a powerful class of nested loops (encountered in partially observable planning) as a set of independent simple loops.

3. Computing Applicability Conditions

We begin by defining the simplest class of abacus programs with loops:

Definition 3. (Simple-Loop Abacus Programs) *A simple loop in a graph is a strongly connected component consisting of exactly one cycle. A* simple-loop *abacus program is one all of whose non-trivial strongly connected components are simple loops.*

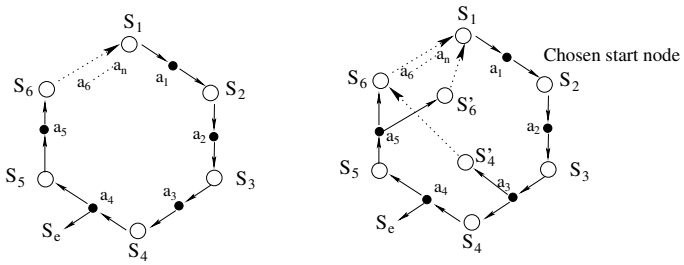

Figure 2: A simple loop with (right) and without (left) shortcuts

We refer to non-trivial strongly connected components that are not simple loops as *complex* loops. In particular, we will be interested in a special class of complex loops, i.e., those obtained by adding "shortcuts" in a simple loop:

Definition 4. (Simple loop with shortcuts) *A simple loop with shortcuts is a strongly connected component $\mathcal{C}$ which includes a node S_0, designated the* start node*, such that removing S_0 makes $\mathcal{C}$ acyclic.*

Intuitively, a simple loop with shortcuts consists of a simple loop with all elements, starting at the start node, in increasing linear order. For any pair of nodes along the loop, a preceeding b, a shortcut from a to b may be added; different shortcuts may overlap as long as this does not create cycles. (e.g., node S_2 can be designated the start node in Fig. 2).

Simple loops with shortcuts form a very general class of complex loops: graph theoretically, this is exactly the class of strongly connected components with *cycle rank* (Eggan 1963) 1. Many control flows that are typically understood as "nested" loops in programming can be represented as simple loops with shortcuts.

The advantage of this class of loops is that we can decompose them into simple loops; in the definition below, a cycle has no repeated nodes other than the start and end nodes.

Definition 5. *(Loop Decomposition) Let $\mathcal{K}$ be a graph in the form of a simple loop with shortcuts with start node S_0. The loop decomposition of $\mathcal{K}$ is defined as the set of all cycles of $\mathcal{K}$ beginning with S_0.*

In the worst case, the size of this decomposition can be exponential in the number of shortcuts. This construction proves useful because in a simple loop with shortcuts, every cycle must contain the start node (this is immediate from Def. 4). Thus, the execution of a simple loop with shortcuts can be viewed as a sequence of complete executions of the simple loops in its decomposition. For instance, we can view the loop with shortcuts in Fig. 2 as consisting of 3 different simple loops. The order of execution of these loops, and whether a particular loop will be executed at all, will depend on the value of registers in states S_3, S_4 and S_5.

We consider a restriction of simple loops with shortcuts in which the preconditions for reaching any state in an abacus program can be determined:

Definition 6. (Monotone simple loops with shortcuts) *A simple loop with shortcuts in an abacus program is monotone iff the sign (positive or negative) of the net change, if any, in a register's value is the same for every simple loop in its decomposition.*

Our main contribution is a method for computing a set of linear constraints between variables representing initial register values, variables representing the number of iterations of each loop, and constants. These conditions determine when a given, initial valuation of registers can lead to a desired state with a desired set of register values. In order to state this result, we make a final definition that determines the accuracy of these conditions:

Definition 7. (Order Independence) *A simple loop with shortcuts is order independent if for every initial valuation of the registers at S_{start}, the set of register-values possible at S_{start} after any number of iterations does not depend on the order in which those iterations are taken.*

Our main result can be summarized as follows:

Theorem 1. *Let Π be an abacus program, all of whose strongly connected components are simple loops with monotone shortcuts.*

Let S be any node in the program, and $\bar{F}$ a vector of register values. We can then compute a disjunction of linear constraints, $\ell(\Pi, S, \bar{F})$, which constitute sufficient conditions on the initial register values for reaching S with the register values $\bar{F}$.

If all simple loops with shortcuts in Π are order independent, then the conditions $\ell(\Pi, S, \bar{F})$ are also necessary.

This result gives us a way of computing the applicability conditions of plans with monotone, simple loops with shortcuts. The accuracy of the computed conditions depends on the notion of order-dependence, which is also crucial in determining whether we can reason about a generalized plan.

4. Applications and Conclusion

The methods discussed above apply very broadly, whenever the effects of planning actions can be represented using increment or decrement operations on some counters. These counters could measure, for instance, changes in the number of objects satisfying some relevant conditions.

We have recently presented an approach for extracting this information from plans in certain domains (Srivastava et al. 2011b). Fig. 3 shows an example plan for the recycling problem, where a recycling agent needs to visit a set of bins and from each bin, collect *paper* and *glass* objects into containers for each kind. Both containers have limited capacities. Both of the collect actions in Fig. 3 decrease the capacity of these containers by one and the edge labeled *empty(b)* is followed when bin *b* is empty (this edge represents a decrement in the number of non-empty bins). Our methods can easily prove that such plans terminate, and that their preconditions assure, for example, that container capacities are larger than the number of objects that need to be recycled.

We have already used these methods successfully to generalize sample classical plans (Srivastava et al. 2011b) and to develop a hybrid approach for constructing generalized plans without any input examples (Srivastava et al. 2011a). The hybrid approach works by incrementally (a) creating an instance of the currently unsolved class of problems, (b) using a classical planner to get a plan for this instance, (c) generalizing this plan, and finally, (d) merging this generalization with the existing generalized plan, which is originally

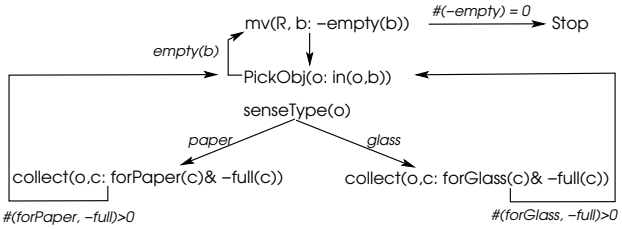

Figure 3: Solution plan for the recycling problem

empty. In this process, we use the approach presented here to create safe and useful loops in steps (c) and (d), and also to identify valid unsolved problem instances.

Our methods can also be applied in various settings where cyclic flows of control need to be constructed. For example, the problem of *programming by demonstration* (PBD) is to construct a program by generalizing input sequences of operations. One of the biggest challenges in PBD is to generalize the demonstrated sequences into *correct* loops of actions and establish their termination conditions. Existing approaches (e.g. (Lau et al. 2003)) often require user-annotations to identify loop iterations and associate confidence scores with the resulting generalizations. Our methods provide two distinct advantages: (a) they can rank different possible generalizations on the basis of semantic measures such as their *domain coverage*, or the fraction of problem instances of interest that they would solve, and (b) they can prune loops that can be proved to be non-terminating, thus improving the space of hypothesized generalizations.

The results we present are also relevant to *workflow inference* (Eker et al. 2009; Yaman and Oates 2007), which is similar to PBD, but emphasizes learning loops of data-processing actions from example traces, and *automated service composition*, where functionalities offerred by different software or web services need to be integrated to achieve new composite services. While various approaches exist to construct linear compositions of services that may internally include cyclic control flows (Bertoli et al. 2010; Narayanan and McIlraith 2002), our approach could provide methods for constructing more powerful, cyclic compositions. For instance, given a website that lists restaurant cuisines, ratings, and addresses along with a public-transportation website, a graduate student attending a conference may want to find all budget restaurants that serve a particular cuisine with easy access from both the conference venue and his or her hotel. A composed service for achieving these results would have to extract an unknown number of qualifying restaurants, and use the transport computation service for each, in a loop.

Finally, our approach is also applicable to *program verification and synthesis*. In this area, our approach is related to methods for proving termination of programs. In particular, the TERMINATOR system (Cook et al. 2006) uses an abstracted formulation of action operators to construct linear *ranking functions*, or functions of the loop variables that are bounded above zero, but *must* decrease in every iteration of the loop. Our approach could be used to address an orthogonal dimension of this problem, to determine the conditions when a program can be guaranteed to reach a goal state, even if it does not always do so.

To Summarize, loop structures play a crucial role in efforts to increase the scope of automated planning and a range of related AI tasks. The methods for computing applicability conditions for plans with loops will thus pave the way for accelerated research in these areas.

References

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. of the 17th Int'l Conf. on Automated Planning and Scheduling*, 26–33.

Bertoli, P.; Pistore, M.; and Traverso, P. 2010. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence* 174(3-4):316–361.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. of the 19th Int'l Conf. on Automated Planning and Scheduling*, 34–41.

Cook, B.; Podelski, A.; and Rybalchenko, A. 2006. Termination proofs for systems code. In *Proc. of the 2006 ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 415–426.

Eggan, L. C. 1963. Transition graphs and the star-height of regular events. *Michigan Mathematical Journal* 10:385–397.

Eker, S.; Lee, T. J.; and Gervasio, M. 2009. Iteration learning by demonstration. In *AAAI 2009 Spring Symposium on Agents that Learn from Human Teachers*.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. Technical Report, AI Center, SRI International. SRI Project 8259.

Hu, Y., and Levesque, H. J. 2010. A correctness result for reasoning about one-dimensional planning problems. In *Proc. of the 12h Int'l Conf. on Principles of Knowledge Representation and Reasoning*.

Lambek, J. 1961. How to program an infinite abacus. *Canadian Mathematical Bulletin* 4(3):295–302.

Lau, T.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53(1-2):111–156.

Levesque, H. J. 2005. Planning with loops. In *Proc. of the 19th Int'l Joint Conf. on Artificial Intelligence*, 509–515.

Minsky, M. L. 1967. *Computation: finite and infinite machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Narayanan, S., and McIlraith, S. A. 2002. Simulation, verification and automated composition of web services. In *Proc. of the 11th Int'l Conf. on World Wide Web*, 77–88.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2010. Computing applicability conditions for plans with loops. In *Proc. of the 20th Int'l Conf. on Automated Planning and Scheduling*, 161–168.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011a. Directed search for generalized plans using classical planners. In *Proc. of the 21st Int'l Conf. on Automated Planning and Scheduling*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011b. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* 175(2):615–647.

Winner, E., and Veloso, M. 2007. LoopDISTILL: Learning domain-specific planners from example plans. In *ICAPS Workshop on AI Planning and Learning*.

Yaman, F., and Oates, T. 2007. Workflow inference: What to do with one example and no semantics. In *AAAI Workshop on Acquiring Planning Knowledge via Demonstration*.