Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Tractability of Planning with Loops

Siddharth Srivastava[1] **Shlomo Zilberstein**[2] **Abhishek Gupta**[1] **Pieter Abbeel**[1] **Stuart Russell**[1]

[1] Computer Science Division, University of California, Berkeley, CA 94720
[2] School of Computer Science, University of Massachusetts, Amherst, MA 01003

Abstract

We create a unified framework for analyzing and synthesizing plans with loops for solving problems with non-deterministic numeric effects and a limited form of partial observability. Three different action models—with deterministic, qualitative non-deterministic and Boolean non-deterministic semantics—are handled using a single abstract representation. We establish the conditions under which the correctness and termination of solutions, represented as abstract policies, can be verified. We also examine the feasibility of learning abstract policies from examples. We demonstrate our techniques on several planning problems and show that they apply to challenging real-world tasks such as doing the laundry with a PR2 robot. These results resolve a number of open questions about planning with loops and facilitate the development of new algorithms and applications.

1 Introduction

The inclusion of loops in plans is crucial for several reasons. First, loops allow compact plans to repeat a sequence of actions, many times, for example in order to produce a sufficient quantity of a certain material. Second, loops facilitate the creation of plans that handle partial observability or non-deterministic actions—both of which introduce uncertainty about the number of times certain actions must be repeated to meet desired conditions. Finally, loops can help increase the applicability conditions of a plan, making it more general and more useful. The utility of loops has motivated a range of planning paradigms with different guarantees of termination and correctness (Cimatti et al. 2003; Levesque 2005; Srivastava, Immerman, and Zilberstein 2008; Bonet, Palacios, and Geffner 2009; Hu and Levesque 2010; Hu and Giacomo 2011).

Consider for example a household robot that needs to do the laundry. Its actions include picking up clothes from a table and a basket; placing clothes in the basket or the washer; picking up the basket and placing it at a location in the laundry room; opening and closing the washer door and moving from the laundry room to the washer room. A state-of-the-art robot with advanced sensors and actuators, such as the PR2, has to use a broad "pinch" grasp with its grippers to pick up the clothes from a heap. The exact number of clothes in the heap and the number that may be picked up with each grasp cannot be determined precisely. Doing the laundry thus represents a challenging planning problem involving partial observability, non-determinism, and unknown quantities of objects. Yet, humans typically consider this task simple enough to be boring. Intuitively, they construct cyclic plans with strong guarantees of correctness and termination: they know that the solution must involve iterations of placing clothes in the basket, carrying the basket to the laundry room, transferring clothes to the washer and so on. Such plans are compact, broadly applicable (regardless of the number of clothes in the initial heap) and surprisingly efficient at dealing with partial observability and non-determinism.

Figure 1: PR2 robot doing the laundry using our planning approach for addressing partial observability and non-determinism.

We examine the feasibility of planning with loops in popular formulations of planning with different action semantics (deterministic or non-deterministic), criteria for correctness (strong cyclic or terminating), and classes of solutions (memoryless or finite-memory abstract policies). We also provide an effective approach for computing plans with loops by learning from examples. First, we characterize precisely different frameworks for planning using action models with deterministic and non-deterministic effects (Sec. 2). We focus on actions that have numeric effects on a set of counter variables. This captures the fundamental aspects of planning with loops, as the counter variables can represent the cardinalities of sets characterized by different properties (Boolean planning problems are a special case with 0/1 counters). Second, we prove novel results categorizing the types of plans with loops that are necessary for expressing solutions based on the properties of the planning framework (Sec. 3). Third, we show that for most of the planning frameworks it is possible to construct plans with loops by learning from examples (Sec. 3.3). Finally we demonstrate the effectiveness of our algorithm for learning from examples on problems from the literature and show that it allows the PR2

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

robot to actually do the laundry (Sec. 4).

2 Problem Formulation

To approach planning with loops under different forms of observability and non-determinism, we use a primitive but powerful representation where all variables are numeric variables with $\mathbb{N}$ or $\mathbb{R}$ as their domains. Let $\mathcal{V}$ be the set of such variables. For each $x \in \mathcal{V}$, we define the *levels* for x, $\ell(x)$, as a finite set of natural numbers. The set of *intervals defined by a set of levels* $\{l_1, \ldots, l_k\}$ is the set $\{[0, l_1), [l_1, l_2), \ldots, [l_k, \infty)\}$, $0 < l_1 < \cdots < l_k < \infty$.

Definition 1. *An **action** consists of a precondition, which maps each variable in $\mathcal{V}$ to a union of intervals defined by the levels for that variable, and a set of action effects,* effects(a). *Each member of* effects(a) *is of the form $\oplus x$ or $\ominus x$, where $x \in vars$;* effects(a) *must include at most one occurrence of each variable in $\mathcal{V}$.*

A *concrete state* in a domain is an assignment that maps each variable in $\mathcal{V}$ to a value in that variable's domain. The value of a variable x in a state s is denoted as $s(x)$.

Example 1. *Consider the settlers domain from the International Planning Competition, modified to include partial observability and non-deterministic effects. The domain involves a mining operation that can produce iron by executing a* `smeltIron` *action that consumes some amounts of* ore *and* coal*. There are three mining actions:* `mineOre` *that produces some ore,* `mineCoal` *that produces some coal, and* `mineBoth` *that produces some amounts of both ore and coal. The actions* `sellCoal` *and* `sellOre` *each reduce the amount of coal and ore respectively, and increase the amount of* wealth*. The state includes values of the variables:$\{ore, coal, iron, wealth\}$, which are non-negative. Preconditions of actions require that numeric variables be in certain ranges, e.g.,* `smeltIron` *requires that the amount of ore is at least ol and coal is at least cl:*

action: `smeltIron`
precondition: $ore \in [ol, \infty) \wedge coal \in [cl, \infty)$
effects: $\{\ominus ore, \ominus coal, \oplus iron\}$

The remaining actions can be similarly represented with the mine actions having no preconditions, and the sell actions requiring some minimal quantity of the resource being sold. Given a start state with non-zero amounts of ore and coal, the goal is to reach a state in which the amount of iron is above a given threshold.

Definition 2. *A **planning domain** $\langle \mathcal{V}, \ell, \mathcal{A} \rangle$ consists of a set of variables $\mathcal{V}$, sets of levels $\ell(x)$ for each $x \in \mathcal{V}$ and a set of actions $\mathcal{A}$ defined using the levels $\ell(x)$ for $x \in \mathcal{V}$.*

Definition 3. *A **planning problem** $\langle \mathcal{D}, s_o, g \rangle$ consists of a planning domain $\mathcal{D} = \langle \mathcal{V}, \ell, \mathcal{A} \rangle$, a concrete state s_0 and a goal mapping g from $V \subseteq \mathcal{V}$ to intervals in $\mathbb{R}$ such that for all $x \in V$, $g(x)$ is an element of the set of intervals $\ell(x)$.*

2.1 Semantics of Action Effects

We consider three interpretations of $\oplus, \ominus$ that correspond to popular frameworks in the literature. We use a common framework to study these variants in a unified manner. In this description, s_1 and s_2 stand for concrete states.

Deterministic semantics $\langle \oplus, \ominus \rangle := \langle +, - \rangle$. $a(s_1)$ is the unique state s_2 such that for every $x \in \mathcal{V}$ if $\oplus x$ is an effect of a, $s_2(x) = s_1(x) + 1$ and if $\ominus x$ is an effect of a then $s_2(x) = max(s_1(x) - 1, 0)$. $s_2(x) = s_1(x)$ for all x that don't occur in effects of a.

Qualitative non-deterministic semantics $\langle \oplus, \ominus \rangle := \langle \uparrow, \downarrow \rangle$. In this case $\oplus x$ and $\ominus x$ increase or decrease x by a non-deterministic amount. Similar actions were introduced without levels by (Srivastava et al. 2011b). The amount of change caused due to each action application can make a variable cross at most one level; on the other hand, in any execution, a finite sequence of $\downarrow x$ (or $\uparrow x$) effects is sufficient to make x cross the next lower level if $x \neq 0$ (or the next higher level, if the interval containing x is not $[l, \infty)$). If x is in its first interval, then a finite sequence of decreases is sufficient to make it zero and subsequent decreases have no effect.

Formally, the effect of $\uparrow x$ is to non-deterministically increase x by δ, where $\delta \in [\epsilon, \delta_{max}]$. $\epsilon > 0$ is a fixed but unknown constant in every execution. δ_{max} is defined as follows. Let $s_1(x) \in [l_j, l_{j+1})$, where $[l_j, l_{j+1})$ is one of the intervals defined by $\ell(x)$. Let l_{j+2} be the next level following l_{j+1} if $l_{j+1} \neq \infty$, and ∞ otherwise. Define $\delta_{max}(x) = l_{j+2} - \epsilon - x$. The δ for decreases is constrained similarly, with the additional restriction that if $x = 0$, it is not changed by $\downarrow x$. For brevity, we henceforth refer to such semantics as qualitative semantics.

Boolean non-deterministic semantics $\langle \oplus, \ominus \rangle := \langle +^?, -^? \rangle$. In this interpretation, $s_2 = a(s_1)$ is the set consisting of states where the effects of a may or may not take place. More precisely, if $\oplus x \in$ *effects*(a) then either $s_2(x) = s_1(x)$ or $s_2(x) = s_1(x) + 1$; if $\ominus x \in$ *effects*(a) then either $s_2(x) = s_1(x)$ or, if $s_1(x) > 0$, $s_2(x) = s_1(x) - 1$.

2.2 Abstract Interpretation of Action Effects

We use an abstract interpretation (Cousot and Cousot 1977) that unifies the action semantics discussed above, and gives directly the possible effects of applying actions on sets of states. An *abstract state* associates each variable with one of the intervals defined by its levels. The set of concrete states represented by an abstract state s, denoted as $\gamma(s)$, is the set of concrete states q that map every variable x to a value $q(x) \in s(x)$. Given a level function ℓ, we denote the unique abstract state that represents a concrete state q as $\tilde{q}_\ell$. We omit the subscript if the meaning is clear from the context.

We use a natural ordering on the intervals induced by the levels for a variable, so that the functions *next(I)* and *previous(I)* return the intervals immediately succeeding and preceding I respectively, with the exception that if I is the first (last) interval, then *previous* (*next*) returns I itself. We consider planning domains where action effects increase or decrease variables. In particular, an *action effect* is of the form $\oplus x$ or $\ominus x$, for $x \in \mathcal{V}$. The *abstract interpretation* $\tilde{a}$ of a is defined as follows. $\tilde{a}(s_1)$ is the set consisting of abstract states s_2 such that if $\oplus x \in$ *effects*(a) then either $s_2(x) = s_1(x)$ or $s_2(x) = next(s_1(x))$; if $\ominus x \in$ *effects*(a), then either $s_2(x) = s_1(x)$ or $s_2(x) = previous(s_1(x))$. This abstract interpretation is *sound*: $\tilde{a}(s_1)$ includes all possible effects under any semantics.

$\mathcal{P}_1$: a strong cyclic policy that may not terminate:
when $(ore < ol \wedge coal < cl \wedge iron < il)$ `mineBoth`
when $(ore < ol \wedge coal \geq cl \wedge iron < il)$ `sellCoal`
when $(ore \geq ol \wedge coal < cl \wedge iron < il)$ `sellOre`
when $(ore \geq ol \wedge coal \geq cl \wedge iron < il)$ `smeltIron`

$\mathcal{P}_2$: a strong cyclic policy that is guaranteed to terminate:
when $(ore < ol \wedge coal < cl \wedge iron < il)$ `mineBoth`
when $(ore < ol \wedge coal \geq cl \wedge iron < il)$ `mineOre`
when $(ore \geq ol \wedge coal < cl \wedge iron < il)$ `mineCoal`
when $(ore \geq ol \wedge coal \geq cl \wedge iron < il)$ `smeltIron`

Figure 2: Two possible plans for the mining problem.

2.3 Solutions to Planning Problems

In this paper, we focus on solutions or policies that map abstract states to actions. Formally, an abstract policy π is a partial mapping from abstract states to actions such that π is defined for all non-goal states that are reachable under π from the initial state s_0. [1]

Let the abstract transition graph of a policy π, denoted ts_π, be the graph of the transition system defined by π. The nodes of ts_π are abstract states; edge $s_1 \xrightarrow{a} s_2$ is in the graph iff $\pi(s_1) = a$ and $s_2 \in \tilde{a}(s_1)$. If the only terminal states in ts_π are goal states, then we say that ts_π is *goal-closed*.

Solution Properties Two criteria define the quality of a solution. Whenever the goal state is reached within a finite number of steps in every execution, from every concrete state c represented by the initial abstract state, the policy is *terminating*. If a policy never leads to an abstract state from which the goal *cannot* be reached by following it, then it is *strong cyclic*. That is, from every abstract state that can be reached by executing a strong cyclic policy, there exists an execution consistent with the policy that leads to the goal. Thus, strong cyclicity is strictly weaker than termination.

Example 2. *Fig. 2 shows two possible solution plans for the mining problem, expressed as abstract policies that map sets of states satisfying certain conditions to actions. The policy $\mathcal{P}_1$ is a strong cyclic solution because from every state that can be reached while executing it, there is a* possible *execution that can lead to the goal: initially, there is a possibility of reaching a state with $ore < ol \wedge coal < cl \wedge iron < il$; from this state, the* `mineBoth` *action could lead to a state with $ore \geq ol \wedge coal \geq cl \wedge iron < il$, allowing the* `smeltIron` *action to be taken. However, this policy can result in a non-terminating execution that repeatedly sells ore and coal and then mines them without reaching the goal.*

The second policy $\mathcal{P}_2$ is a strong cyclic policy that is also guaranteed *to terminate under deterministic and qualitative semantics. It is easy to see that in this case the first three mining operations will take place until they reach a state satisfying $ore \geq ol \wedge coal \geq cl \wedge iron < il$, allowing the* `smeltIron` *action to be taken. Clearly, $\mathcal{P}_2$ is preferable over $\mathcal{P}_1$ because $\mathcal{P}_1$ is not guaranteed to reach the goal in a finite number of steps under any semantics.*

[1] *Concrete* policies that map *concrete states* to actions require a compact representation (see Example 4) because the number of concrete states can be uncountably infinite. Unless otherwise specified, the term "policy" refers to an abstract policy.

Algorithm 1: (Progress-Sieve) abstract policy termination test

Input: $g = ts(\pi, \tilde{s_I})$
Remove all edges e of g that have a progress variable w.r.t. their SCCs
if *no edge was removed* **then**
  Return "Non-terminating"
for $g' \in$ SCCs-of(g) **do**
  if *Progress-Sieve(g')= "Non-terminating"* **then**
    Return "Non-terminating"
Return "Terminating"

3 Properties and Computability of Solutions

In this section we investigate fundamental questions about popular planning frameworks with different action semantics, observability conditions and solution requirements. We first investigate the decidability of each of the solution properties and provide algorithms when possible. Under Boolean-ND semantics, termination of cyclic abstract policies can never be guaranteed. Given any cycle in the transition graph, action outcomes can be selected so that the cycle is executed indefinitely. Moreover, cycles must exist in the transition graph because it is possible for an action to have no effect. Thus, terminating policies for such semantics are not possible. We now focus on the remaining frameworks and their solution properties.

Srivastava et al. (2011b) presented the Sieve algorithm for determining if an abstract policy terminates under qualitative semantics for the case where each variable has only one level (0). The Progress-Sieve algorithm (Alg. 1) extends that algorithm to variables with multiple levels by using a generalized notion of progress variables. Like the Sieve algorithm it recurs over strongly connected components (SCCs), and is shown as Alg. 1.

Definition 4. (Progress variable) *Given a policy π, variable x_i is a progress variable w.r.t. a subgraph G of ts_π iff either all occurrences of x_i in the effects of actions in G are $\ominus x_i$ and for every state in G, $x_i \notin [0, l_1)$, or all occurrences of x_i are $\oplus x_i$ and for every state in G, $x_i \notin [l_j, \infty)$.*

It turns out that the Progress-Sieve algorithm with levels is also complete for the qualitative semantics. The inclusion of other levels makes our situation significantly different from prior work due to two reasons: (1) increase effects may also lead to termination of the execution of an SCC; and (2) the single 0 level allows a stronger argument where ϵ, the lower bound on qualitative action effects, can be arbitrary. This argument does not hold in our setting.

Theorem 1. *The Progress-Sieve algorithm is a sound test of termination w.r.t. deterministic and qualitative semantics.*

The proof follows the inductive argument made by (Srivastava et al. 2011b) with the additional observation that in any ϵ bounded trajectory, any variable that is not in its last interval (bounded by ∞) that is repeatedly increased without a decrease must eventually cross a level. Similarly, any variable that is not in its first interval and is repeatedly decreased must eventually cross a level.

Lemma 1. *In any abstract transition sequence with common initial and final abstract states of the form* $s_1 \xrightarrow{\tilde{a}_1} \cdots s_k \xrightarrow{\tilde{a}_k} s_1$*, if no variable is a progress variable then for every* x_i *and every initial valuation* $x_i^0 \in s_1(x_i)$*, there exists* $\epsilon > 0$ *and* $\delta_i^1, \ldots \delta_i^k > \epsilon$ *such that each* δ_i^j *is in the range of the effect of* a_j *using qualitative semantics and* $\sum_{j=1\ldots k} \delta_i^j = 0$*. In other words, the final value* $x_i^k = x_i^0 + \sum_{j=1\ldots k} \delta_i^j = x_i^0$*.*

Proof. Consider sequence of effects on a variable x_1 that is affected by actions in the given sequence. We use induction on k, the number of actions in the sequence. $k = 1$ is a special case. The only way in which a cycle with one action can have no progress variables is if all affected variables are in maximal ($[l_k, \infty)$) or minimal ($[0, l_1)$) intervals and are respectively increased or decreased. In that case, by the definition of qualitative action semantics, the cycle must be non-terminating. $k = 1$ is not valid because every variable will be a progress variable for a path with one action.

If $k = 2$, the last action simply "undoes" the effect of the first action. Suppose the hypothesis holds for $k = m$. Consider a sequence of $m + 1$ actions. We have 2 cases:

Case 1 The last $(m+1)$'th action causes a transition across intervals. In this case, a δ can be chosen for the last action to reach any point in the interval in which x_1^0 belongs.

Case 2 The last action causes a transition within the interval in which x_1^0 belongs. By the induction hypothesis, the sequence of first m actions can be instantiated so that $x_1^0 = x_1^m$. Let the ϵ for which this equality is achieved be ϵ_1.

If the last two actions affect x_1 in the same direction, set $\epsilon_2 < min(\epsilon_1, \delta/2)$ where δ is the affect of the m'th action on x, and let the affect due to the last action be $\delta/2$.

If the last two actions affect x_1 in opposite directions, then change the effect of the m'th action to reach some point $y \neq x_1^0$ such that $y > x_1^0$ (or $y < x_1^1$) if the $(m + 1)$'th action decreases (or increases) x. Let $\epsilon_2 = min(\epsilon_1, |y - x_1^0|)$

In this manner, for any given sequence of $m + 1$ actions, we can compute an ϵ and a sequence of δ_i^j's representing changes on each variable by each action in the sequence such that ϵ is a lower bound on all δ_i^j. □

Theorem 2. *The Progress-Sieve algorithm is a complete test of termination w.r.t. qualitative semantics.*

Proof. Suppose that the Progress-Sieve algorithm returns "non-terminating". This implies the presence of an SCC without a progress variable. Then we can construct a sequence of states and actions $\mathcal{S} = s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_k} s_1$, possibly with repetition such that every action in the SCC appears at least once in this sequence. By Lemma 1, we can create a non-terminating instance of this sequence, so that if the starting value of each variable is in the interior of its interval then after every iteration of the action sequence, the variable values return to the original values. This sequence of actions can be executed ad infinitum. Thus, there is in fact a non-terminating execution sequence. □

Theorem 3. *The Progress-Sieve algorithm is* not *a complete test of termination w.r.t deterministic semantics.*

Proof. The following is a counter-example to completeness. Consider a planning problem with $\mathcal{V} = \{x, y, z\}$ $\ell(x) = \{1, 5\}$, $\ell(z) = \ell(y) = \{1\}$. Let π_1 be defined as: $\pi_1(s_1 = x \in [1, 5), y \in [0, 1), z \in [0, 1)) : +x, +y$
$\pi_1(s_2 = x \in [1, 5), y \in [1, \infty), z \in [0, 1)) : +x, +z$
$\pi_1(s_3 = x \in [1, 5), y \in [1, \infty), z \in [1, \infty)) : -x, -y, -z$
$ts_{\pi 1}$ has the following SCC:
$s_1 \xrightarrow{+x,+y} s_2 \xrightarrow{+x,+z} s_3 \xrightarrow{-x,-y,-z} s_1$ (each state also has a self loop). This SCC has no progress variable but it terminates after at most four steps because x is incremented by 1 in every full execution of the cycle s_1, s_2, s_3. □

Thus, the progress-sieve algorithm is an accurate test for abstract policies under qualitative semantics, but it is only a sound test under deterministic semantics.

We have thus established that for determining that a solution terminates, there is a sound and complete algorithm under qualitative semantics, a sound algorithm under deterministic semantics, and the problem is meaningless for Boolean-ND semantics. For determining strong cyclicity of an abstract policy under any semantics, we only need to check the connectivity properties of the finite abstract transition graph.

The next question is whether it is possible to construct a complete test of termination under deterministic semantics. The following result answers it negatively. Essentially, even abstract policies can be used to represent *abacus programs* (Lambek 1961; Boolos and Jeffrey 1987; Helmert 2002), which are equivalent to Turing machines. Determining if a policy terminates under deterministic semantics is thus equivalent to the halting problem for Turing machines.

Definition 5. (Abacus Programs) *An abacus program* $\langle \mathcal{R}, \mathcal{Q}, \ell, q_0, q_f \rangle$ *consists of a finite set of registers* $\mathcal{R}$*, a finite set of states* $\mathcal{S}$ *with special initial and halting states* $q_0, q_f \in \mathcal{S}$ *and state labels determined by* $\ell : \mathcal{S} \setminus \{q_f\} \mapsto$ Act*. The set of actions,* Act*, includes, for each register* r *and states* q_1, q_2*:*

- $Inc(r, q_1)$*: increment* r*; goto* q_1
- $Dec(r, q_1, q_2)$*: if* $r = 0$ *goto* q_1 *else decrement* r*; goto* q_2

The execution of abacus programs starts at q_0; at every step, the applicable edge leading out of the state is taken and the corresponding action's effect is applied to the relevant register. Execution stops when the state q_h is reached. Abacus programs can be represented as graphs with nodes representing states and edge labels representing actions. Thus nodes for states that are mapped to decrementing actions have two outgoing edges.

Theorem 4. *For any abacus program* $A = \langle \mathcal{R}, \mathcal{Q}, \ell, q_0, q_f \rangle$*, there exists a policy* $\pi(A)$*, and a set of abstract states* S_f *such that an execution of* A *terminates at* q_f *iff the execution of* $\pi(A)$ *terminates at a state in* S_f*.*

Proof. The set of variables over which $\pi(A)$ is defined is $\mathcal{R} \cup \mathcal{Q} \cup \{q_{ij} : q_i, q_j \in \mathcal{Q}\}$. All variables have a singleton level set, $\{1\}$. Let $q_i \in \mathcal{Q}$ and $a_{r,i,j}$ be the action that effects r and leads to q_j. Let the effect of $a_{r,i,j}$ along the edge from q_i to q_j be *eff*$(a_{r,i,j})$. to a state q_j, the policy $\pi(A)$ includes the following mappings. For ease in presentation, we only write the state variables that are not in $[0, 1)$ on the left.

	Deterministic	Qualitative	Boolean-ND
Strong cyclic	N/A	True	False
Terminating	False	True	N/A

Table 1: Sufficiency of abstract solution policies for fully observable problems.

	Deterministic	Qualitative	Boolean-ND
Strong cyclic	True	True	True
Terminating	False	True	N/A

Table 2: Sufficiency of memoryless abstract solution policies for partially observable problems.

$q_i \in [1,\infty)$: $+q_{ij}, \mathit{eff}(a_{r,i,j})$
$q_i \in [1,\infty), q_{ij} \in [1,\infty)$: $-q_i$
$q_{ij} \in [1,\infty)$: $+q_j$
$q_j \in [1,\infty), q_{ij} \in [1,\infty)$: $-q_{ij}$

In the initial state, all variables other than q_0 are set to $[0,1)$; q_0 is set to $[1,\infty)$. With this construction, execution of an abacus program is captured by the execution of $\pi(A)$ in the sense that the state variable not in $[0,1)$ denotes the "current" execution state of the abacus program. If the edge from q_i to q_j is to be taken in the abacus program, the policy increments q_{ij}, decrements q_i, increments q_j and finally decrements q_{ij} to achieve the representation of being in the abacus program state q_j. The intermediate variables $q_{i,j}$ are required because we cannot set q_0, to be exactly 1 initially and must allow for multiple decrements of q_0 to bring it to 0. Thus, the execution of the policy corresponds to the execution of the given abacus program and the policy terminates at a state with $q_f \in [1,\infty)$ and the remaining state variables at $[0,1)$. □

3.1 Sufficiency of Policy Classes for FO Problems

The following sections describe the requirements for expressing solutions to planning problems under full observability (FO) and partial observability (PO), under all the possible combinations of action semantics and solution properties. We show that it is not always possible to express a solution as an abstract policy, but sometimes the use of a finite amount of memory with an abstract policy can be sufficient. All the results are summarized in Tables 1 and 2. Finally, we show that a large class of problems can be solved by learning from examples.

Sufficiency of Abstract Terminating Policies We first investigate if the existence of a terminating concrete solution implies the existence of a terminating abstract solution.

Example 3. *Let domain $\mathcal{D}_1$ be defined using the variables $\{x, y, z, w\}$. x, y, w have the level $\{1\}$ while z has the level $\{5\}$. The set of actions, expressed as tuples of preconditions and effects, is as follows: $a_1 = \langle (x, y \in [0,1)), (\oplus y, \oplus z)\rangle$, $a_2 = \langle (x, y \in [0,1)), (\oplus x, \oplus z)\rangle$, $goalA_1 = \langle (y \in [1,\infty), z \in [5,\infty)), (\oplus w)\rangle$, $goalA_2 = \langle (x \in [1,\infty), z \in [0,5)), (\oplus w)\rangle$. The goal condition is $w \in [1,\infty)$.*

Under deterministic semantics, one of the two actions $goalA_1$ or $goalA_2$ need to be applied to reach the goal. If $z < 4$ initially, applying a_2 enables the application of $goalA_2$ but not $goalA_1$. If $z \geq 4$, applying a_1 enables the application of $goalA_1$ but not $goalA_2$.

Example 4. *A concrete policy π_{conc} for Eg. 3 can be specified compactly as follows: $z \in [4,\infty) : a_1$; $z \in [0,4) : a_2$; $y \in [1,\infty), z \in [5,\infty) : goalA_1$; $x \in [1,\infty), z \in [0,5) : goalA_2$. This is not an abstract policy because it uses the level 4 for z.*

Theorem 5. *Under deterministic interpretation of $\langle \oplus, \ominus \rangle$, existence of a concrete terminating, goal-closed policy for a fully observable planning problem does not imply the existence of an abstract terminating, goal-closed policy.*

Proof. A counterexample is obtained using the domain $\mathcal{D}_1$ (Eg. 3) with the planning problem obtained by setting the concrete initial state c_0 as $x = y = w = 0$, $z = 3$. π_{conc} (Eg. 4) is a terminating, goal-closed policy under deterministic semantics, *for any initial value of z*. However there is no abstract, terminating goal-closed policy. Consider the initial abstract state $s_0 = x, y, w \in [0,1); z \in [0,5)$. If a policy sets a_1 for s_0, one of the possible outcomes is $x \in [0,1), y \in [1,\infty), z \in [0,5)$, in which no action is applicable. If a_2 is assigned to s_0, the possible result $x \in [1,\infty), y \in [0,1), z \in [5,\infty)$ has no applicable action. Therefore, this problem has no abstract solution policy. □

The following lemma and theorem follow from existing results. Since their proof applies directly to our setting, we only state them here for completeness. The set of abstract trajectories of a given policy is the set of sequences $s_0, s_1, \ldots$ where $s_i \neq s_j$, obtained by executing the policy using abstract interpretations of actions.

Lemma 2. *(Srivastava et al. 2011b) Let c_1 and c_2 be abstract states in a domain $\mathcal{D}$. If $\tilde{c}_1 = \tilde{c}_2$ then c_1 and c_2 have the same set of abstracted trajectories w.r.t. any concrete policy π.*

Theorem 6. *(Srivastava et al. 2011b) Under qualitative semantics, a concrete, terminating solution policy exists for a fully observable planning problem iff an abstract solution policy does.*

As noted earlier, terminating concrete policies for Boolean-ND semantics do not exist.

Sufficiency of Abstract Strong Cyclic Policies Before we determine whether abstract policies are sufficient when we are only interested in strong cyclic policies, we clarify a few key notions. We say that a concrete policy is cyclic if it is possible to start executing the policy and visit the same concrete state at least twice. A concrete policy is strong cyclic if the policy is cyclic, and every state visited during the execution has an execution trajectory consistent with the policy, leading to the goal. Problems with deterministic action semantics cannot have strong cyclic policies because if the policy is cyclic, the states that are in a cycle will have no paths to the goal since actions will have unique outcomes.

The following result follows from arguments presented above for Thm. 6.

Theorem 7. *Under qualitative semantics, a concrete strong cyclic solution policy exists for a fully observable planning problem iff an abstract strong cyclic solution policy does.*

Example 5. *Consider a planning domain $\mathcal{D}_2$ with variables x and y. The levels for x and y respectively are $\{1, 3\}$ and $\{1, 5\}$ respectively. The set of actions includes $a_1 = \langle (x \in [0, 1)), (\oplus x) \rangle; a_2 = \langle (x \in [1, 3)), (\oplus x, \oplus y) \rangle; a_3 = \langle (x \in [1, 3)), (\ominus x, \oplus y) \rangle$.*

Example 6. *Let $\mathcal{D}'_2$ be the domain obtained by removing action a_1 in $\mathcal{D}_2$ (defined in Eg. 5). Define a fully observable planning problem P_2 over $\mathcal{D}'_2$ with the start state $x = 1, y = 0$ and the goal condition $y \in [5, \infty)$. We can specify a concrete partial policy, π_{conc2} in this domain as $x = 1, y \in [0, 5) : a_2; x = 2, y \in [0, 5) : a_3$. Under Boolean-ND semantics, an execution of the policy starting with the initial state will apply a_2 until either x is increased, or y, or both. If in the process y reaches 5, execution stops. If x reaches 2, application of a_3 is repeated until either x becomes 1 (and application of a_2 resumes again) or y becomes 5. In either case, there is a path to reaching the goal state and thus the policy is strong cyclic.*

Theorem 8. *Under Boolean-ND semantics, existence of a concrete strong cyclic solution policy for a fully observable planning problem does not imply existence of an abstract strong cyclic solution policy.*

Proof. Eg. 6 presents a concrete strong cyclic policy for $\mathcal{D}'_2$ in Eg. 5. However, there is no abstract strong cyclic policy because the result of applying $\tilde{a}_2$ on any abstract state with $x \in [1, 3)$ in that example has as its possible outcomes, abstract states with $x \in [3, \infty)$, with no path to the goal. □

3.2 Sufficiency of Policy Classes for PO Problems

So far we discussed the sufficiency of abstract solution policies while defining solvability as the existence of a concrete solution policy. However, in many real-world situations the state is only partially observable. In such situations the agent may not know the value of each state variable precisely, and thus cannot execute concrete solution policies that map arbitrary values or intervals of variables to actions. We consider a form of partial observability where the agent only knows the interval that each variable belongs to, where the intervals are fixed per problem. That is, the set of observable intervals for each variable is the set of intervals defined by its levels.

Definition 6. *A **partially observable planning problem** $\langle \mathcal{D}, s_o, g \rangle$ consists of a planning domain $\mathcal{D} = \langle \mathcal{V}, \ell, \mathcal{A} \rangle$, an* abstract state *$s_0$ in $\mathcal{D}$, and g, a mapping from $V \subseteq \mathcal{V}$ to intervals in $\mathbb{R}$ such that for all $x \in V$, $g(x)$ is an element of the set of intervals defined by $\ell(x)$.*

Abstract policies satisfy the epistemic constraints imposed on the agent in such problems but, as we show below, are in some cases insufficient compared to "finite-memory abstract policies" that can incorporate a finite history of executions and observations. A finite-memory abstract policy is defined in the form of a finite-state controller. As expected, the definition reduces to that of an abstract policy when the set of memory states is empty.

Definition 7. (Finite-memory policy) *A finite-memory abstract policy $\langle \mathcal{Q}, q_0, \pi \rangle$ over a domain $\mathcal{D} = \langle \mathcal{V}, \ell, \mathcal{A} \rangle$ consists of a finite set of memory states $\mathcal{Q}$, an initial memory state $q_0 \in \mathcal{Q}$, and a mapping $\pi : \mathcal{Q} \times S \mapsto \mathcal{Q} \times \mathcal{A}$, where S is the set of abstract states defined by $\mathcal{D}$.*

As in memoryless policies, a finite-memory policy may be partial, covering only the reachable states. A finite-memory policy is *goal-closed* if the only terminal states (defined as pairs of memory states and abstract states) in the abstract transition system induced by the policy are goal states.

Sufficiency of Memoryless Strong Cyclic Policies The following result shows that memoryless abstract policies are actually sufficient when strong cyclic policies are required.

Theorem 9. *Suppose π is a finite-memory strong cyclic policy w.r.t. the goal condition g. Then there exists a policy π' which is also strong cyclic w.r.t. g, but does not use memory.*

Proof. We construct the policy π' as follows. Consider the transition graph ts_π, whose nodes represent dual states defined as pairs of a memory state and a problem state. Define ts'_π as the graph obtained by merging all nodes with the same problem state. Nodes of ts'_π are labeled only with problem states and can have multiple outgoing edges. Repeat the following process for every goal state s_g. Let $border(k)$ be the set of states whose min distance in terms of actions from the goal state is k. Thus $border(0)$ = goal state. Iterate over non-empty sets $border(k)$, in increasing order of k, the following operation: for each state in $border(k)$, remove outgoing edges corresponding to all actions except those for an action that leads to a state in $border(k-1)$.

After every step of this process, every state in ts'_π has a path to the goal. We prove this by induction on k. The claim is true initially because the policy ts_π was strong cyclic. After the $k = m$ iteration, let s be a state in $border(m+1)$. The pruning is such that s still has a path to some state in $border(m)$. By induction, that state has a path to the goal and therefore s has a path to the goal. Thus, π' is strong cyclic. □

Sufficiency of Memoryless Terminating Policies We now show that if termination is required, the sufficiency of *memoryless* policies depends on the semantics.

Example 7. *In this example we construct a finite memory terminating policy for the domain $\mathcal{D}_2$ described in Eg. 5. We first define a partially observable planning problem P_3 with the initial state $x \in [0, 1), y \in [1, 5)$ and the goal condition $y \in [5, \infty)$. Define the finite-memory policy π_{conc2} using two memory states q_0, q_1 as follows. $\langle q_0, (x \in [0, 1), y \in [0, 1)) \rangle : \langle q_0, a_1 \rangle; \langle q_0, (x \in [1, 3), y \in [1, 5)) \rangle : \langle q_1, a_2 \rangle; \langle q_1, (x \in [1, 3), y \in [1, 5)) \rangle : \langle q_0, a_3 \rangle$. Under deterministic semantics, an execution of this policy results in a finite action application sequence of the form $a_1, a_2, a_3, a_2, a_3, \ldots$. The execution terminates only when y reaches 5. π_{conc2} is therefore a finite-memory, terminating, goal-closed policy.*

Theorem 10. *Under deterministic interpretation of $\langle \oplus, \ominus \rangle$, existence of a finite-memory terminating, goal-closed policy for a partially observable planning problem does not necessarily imply the existence of a memoryless terminating, goal-closed policy.*

Proof. Eg. 7 gives the required counterexample. This problem has no memoryless abstract policy because the same abstract state, $x \in [1, 3), y \in [1, 5)$ requires a finite number of repetitions of the action sequence a_1, a_3. □

Theorem 11. *Under qualitative interpretation of $\langle \oplus, \ominus \rangle$, a finite-memory terminating, goal-closed policy for a partially observable planning problem exists iff a memoryless terminating, goal-closed policy does.*

Proof. Consider the fully observable version of the given problem. If this problem has a terminating solution, it must have a memoryless concrete policy as a solution (adding memory in fully observable problems has no benefit). If it does, then by Thm 6 it must have a qualitative solution policy, which also solves the partially observable problem. In other words the partially observable planning problem has a terminating solution policy iff the fully observable problem does and the fully observable planning problem has a terminating solution policy iff it has an abstract terminating solution policy. □

3.3 Feasibility of Learning from Examples

The preceding sections show that memoryless abstract policies are sufficient for six of the nine meaningful combinations of action semantics, solution criteria, and observability (Tables 1 and 2). With this motivation, we consider the problem of efficiently computing abstract policies. One approach is to enumerate all possible abstract policies, and select one that is terminating and goal-closed (Srivastava et al. 2011b). Clearly, this approach is not scalable. Learning from examples would be possible only if every desirable policy could in principle be constructed using a finite set of example plans. Ideally, it should also be possible to generate these plans by invoking planners with a well defined goal condition. We show the surprising result that both these conditions hold for abstract memoryless policies.

Given a concrete state c_0 and a sequence of actions $p = p_1, \ldots, p_n$ the application of p on c_0 with *deterministic semantics* ($+1, -1$ for $\oplus, \ominus$) is denoted as $p(c_0)$. $p(c_0)$ is a sequence of concrete states $c_o, c_1, \ldots, c_n$. If this sequence satisfies $\tilde{c}_k = \tilde{c}_l \implies p_l = p_k$ under a level mapping ℓ, then we say that p *respects* ℓ *when executed on* c. Such a plan naturally defines a partial policy on the states $\tilde{c_0}, \ldots \tilde{c}_n$: $\pi(\tilde{c}_l) = p_l$. Consider a finite set of example plans $\langle c^i, p^i \rangle$ such that p^i respects ℓ when executed on c^i. We define such a set to be consistent with ℓ if whenever two concrete states c^i_l, c^j_k in any two executions $p^i(c^i)$ and $p^j(c^j)$ respectively are such that $\tilde{c}^i_l = \tilde{c}^j_k$, we have $p^i_k = p^i_l$. Using this notation, the desired result is as follows.

Theorem 12. *Given a level mapping ℓ and memoryless abstract policy π which is goal-closed and terminating, there exists a finite set of goal-achieving example plans which use deterministic semantics ($x+1, max(0, x-1)$ for $\oplus, \ominus$), are consistent with ℓ, and collectively define π.*

Proof. We construct the set of examples as follows. Unmark all states in ts_π. Repeat the following steps until all states are marked: (1) Select an unmarked node s. Create a concrete state $c \in \gamma(s)$, with variable values in $\mathbb{N}$. Apply policy π starting at c, using deterministic semantics; (2) Mark all states reached during this process. Include the trajectory executed starting from c in the set of examples. Every step of the application in step 1 results in a transition in the concrete state space. Since our abstract interpretation is sound, every successive concrete state must belong to one of the resulting abstract states included in π. Since π is terminating (termination under qualitative semantics implies termination under deterministic semantics because effects under deterministic semantics are subsumed by those under qualitative semantics), this execution must end after a finite number of steps by reaching a state not in the domain of the policy. Since π is goal-closed, this state must be a goal state.

Every iteration of this procedure marks at least one additional state in ts_π, so it must terminate. The set of examples constructed while executing step 2 is the desired set. □

4 Implementation and Validation

Thm. 12 suggests that our desired solutions can be constructed by combining solutions to appropriate deterministic planning problems. With this motivation, we implemented a version of the hybrid search algorithm presented by (Srivastava et al. 2011a). This algorithm incrementally generates deterministic concrete planning problems and solves them using a classical planner. The concrete solutions are simulated using the abstract action semantics developed in this paper, resulting in an abstract transition graph. The algorithm successively creates concrete problem instances (with deterministic semantics) corresponding to non-terminal non-goal states in this graph, repeats the entire abstraction process and merges the new transition graph with the existing graph. In doing so, it creates copies of abstract states in the graph when needed to ensure termination. The implementation is in Java and uses FF (Hoffmann and Nebel 2001; Hoffmann 2003) as the classical planner.

Problems from the literature We experimented with versions of the mining problem from the settlers domain by adding various items to the production chain (e.g., tools can be produced when there is a sufficient amount of iron), as well as the problems considered by (Srivastava et al. 2011b) and those by (Bonet, Palacios, and Geffner 2009) that have memoryless solutions. We summarize the results below:

- *Timing* All the problems were solved in at most $1s$ (on a 1.7GHz Intel Core i5 Mac), including the time for generating and solving concrete problems.
- *Number of examples* The average numbers (over 10 runs) of concrete plans generated were: *delivery+fuel*: 4.4; *snow*: 1.2; *trash-collection*: 6; *tree*: 1; *nestedVar*: 1; *mining* (producing tools): 4.1; *corner* and *hall*: 1.

The difficulty of *nestedVar* can be scaled by increasing the number of variables. With 10 variables, the number of policies is $10^{2^{10}}$ with only one true solution. This makes search infeasible, but our approach solves the problem in 2.2s by generating and solving just 1 problem instance. The solutions computed using our approach are also robust in that they are terminating solutions under qualitative and deterministic semantics of actions, and strong cyclic for possibilistic semantics. In addition, levels can be scaled arbitrarily without affecting the solution properties.

Real-World Validation with the PR2 Robot In order to demonstrate the practical validity of our approach, we also applied it to a real-world task using the PR2 robot. Our system computed high-level terminating plans with loops by

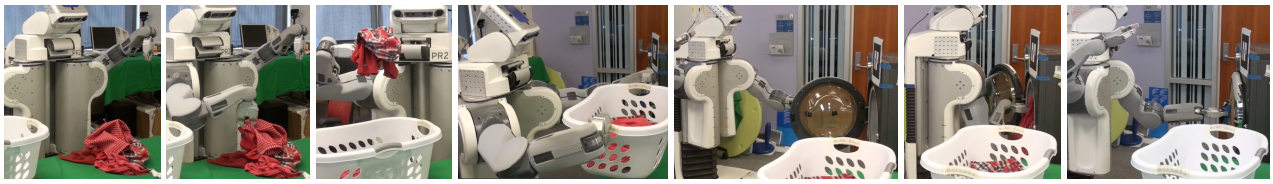

Figure 3: Snapshots of the PR2 doing laundry using our approach.

generalizing and merging example plans using abstract semantics as described in Sec. 4. These high-level plans were refined into motion plans by extending a recently developed approach for combined task and motion planning for fully observable deterministic problems (Srivastava et al. 2014). In order to refine high-level plans with loops, at every stage of execution, we refine and execute only the segment of the plan up to the next branch, followed by a sensing action for resolving that branch. E.g., while loading the basket, the policy includes a branch on the number of dirty clothes on the table. Instead of generating motion plans for all the possible levels of this count, we execute the detect-clothes-ontable action and refine only the branch whose condition is satisfied by the detected result. We used a Kinect sensor to obtain RGBD data and OpenCV's contour generation based on color masks to detect clothes as objects with a specific range of colors (close to red). In this implementation, the physical pose at which a detection is required for resolving the level of a variable (such as *num_dirty_clothes_on_table*) is derived from variable names using a lookup table for the poses corresponding to surfaces and other fixed locations.

We validated this approach with the laundry task (Fig. 3). In this problem, the state is defined by a combination of Boolean and Natural number variables including the number of clothes at the table, the basket and the washer; whether or not a gripper is free; whether or not the robot is holding the basket, etc. The initial state has a heap with an unknown number of clothes on the table in a closet and a washer is present with its door closed. As noted in the introduction, the available actions are to move between various locations, pick up and place objects (such as clothes and a laundry basket) at various locations (into the washer, the basket, at the table with dirty clothes, or at a location in the laundry room). The solution plan involves iteratively picking up the clothes and placing them into the basket; moving from the "closet" where the dirty clothes are, to the "laundry room"; placing the basket at a table in the laundry room; opening the washer; incrementally loading the clothes into the washer and finally, closing the washer door. Note that the laundry basket is available for transporting clothes, but the robot can also use its grippers and make multiple trips for doing so. Our system computes the solution plan (that uses the basket) in less than 1 second. A video of *the PR2 robot doing laundry* using this approach is available at `http://tiny.cc/laundrybot`.

5 Discussion and Conclusions

We developed a unified framework that captures several popular planning paradigms where plans with loops are useful. It allowed us to prove fundamental results and facilitated an effective solution approach for constructing robust plans with loops in real-world scenarios. Our comprehensive analysis also sheds light on the limits of feasibility, which can make future algorithm development efforts more effective.

Strong cyclic policies have been studied extensively for several applications including the design of agent behaviors and automated service composition (Bertoli, Pistore, and Traverso 2010). Another popular direction of research focuses on computing cyclic controllers that are guaranteed to work for only one problem instance with a finite state space (Bonet, Palacios, and Geffner 2009; Hu and De Giacomo 2013). These approaches leverage the succinctness of plans with loops, but are limited in their ability to exploit the broad applicability afforded by cyclic plans. While they provide solutions that could work for multiple problem instances, this is only a post-facto observation. They are only guaranteed to solve a single instance with a finite state space, and they cannot produce a characterization of the general class they may solve. In contrast, our results apply directly to generalized planning (Srivastava, Immerman, and Zilberstein 2011), where the agent has to solve sets of infinitely many planning problems with unknown numbers of objects. Solutions computed by our approach are *guaranteed* to solve instances with different variable values that can represent unbounded counts of objects (e.g., the solution for laundry is guaranteed to terminate and solve the problem for any finite but unbounded number of dirty clothes; each value of this quantity constitutes a distinct planning problem with a finite state space).

The algorithm presented in Sec. 4 draws upon prior work on incrementally obtaining example plans, generalizing them using abstract interpretation, and merging them while ensuring goal reachability and termination. However, in order to use this process under the much broader class of action semantics and problem formulations presented here, the components for action application, abstract interpretation and termination analysis require the new approach developed in this paper. As a result of using abstract semantics, the computed solutions are simultaneously strong (terminate at a goal state) under deterministic and qualitative semantics *and* strong cyclic under Boolean-ND semantics.

The Progress-Sieve algorithm is more general than existing algorithms for determining termination (Hu and Levesque 2010; Srivastava et al. 2011b), even for the special case of deterministic action effects. Numeric planning has reached a mature level of development (Coles and Coles 2011; Helmert 2002), but existing approaches do not address the inclusion of loops in plans, partial observability and nondeterministic effects.

Acknowledgments

This research was supported in part by the ONR under grant N00014-12-1-0609 and by the DARPA Young Faculty Award #D13AP00046.

References

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

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 International Conference on Automated Planning and Scheduling*, 34–41.

Boolos, G., and Jeffrey, R. C. 1987. *Computability and Logic (2nd ed.)*. Cambridge University Press.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2):35–84.

Coles, A. J., and Coles, A. I. 2011. LPRPG-P: Relaxed plan heuristics for planning with preferences. In *Proc. of the 21st International Conference on Automated Planning and Scheduling*, 26–33.

Cousot, P., and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 238–252.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *Proc. of the 6th International Conference on Artificial Intelligence Planning and Scheduling*, 44–53.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2003. The metric-FF planning system: Translating "ignoring delete lists" to numerical state variables. *Journal of Artificial Intelligence Research. Special issue on the 3rd International Planning Competition* 291–341.

Hu, Y., and De Giacomo, G. 2013. A generic technique for synthesizing bounded finite-state controllers. In *Proc. of the 23rd International Conference on Automated Planning and Scheduling*, 109–116.

Hu, Y., and Giacomo, G. D. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence*, 918–923.

Hu, Y., and Levesque, H. J. 2010. A correctness result for reasoning about one-dimensional planning problems. In *Proc. of the 12th International Conference on Principles of Knowledge Representation and Reasoning*, 362–371.

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

Levesque, H. J. 2005. Planning with loops. In *Proc. of the 19th International Joint Conference on Artificial Intelligence*, 509–515.

Srivastava, S.; Immerman, N.; Zilberstein, S.; and Zhang, T. 2011a. Directed search for generalized plans using classical planners. In *Proc. of the 21st International Conference on Automated Planning and Scheduling*, 226–233.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011b. Qualitative numeric planning. In *Proc. of the 25th Conference on Artificial Intelligence*, 1010–1016.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In *Proc. of the IEEE International Conference on Robotics and Automation*, 639–646.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proc. of the 23rd National Conference on Artificial Intelligence*, 991–997.

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