Planning with Loops: Some New Results

Yuxiao Hu and **Hector J. Levesque**
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{yuxiao, hector} @cs.toronto.edu

Abstract

In AI planning, there has been an increasing interest in solving a *class* of problems, rather than individual problems, with a generalized notion of "plan." One such generalization is plans with loops, *i.e.* program-like plans, whose execution on a specific problem in the class results in a sequential plan. Levesque's KPLANNER falls into this paradigm: it generates robot programs, a type of loopy plan, that solves a range of parametrized planning problems. In this paper, we build on that work, and propose another plan representation along with a novel planning algorithm. We show that the new plan representation is more general than robot programs, and the new planner more efficient than KPLANNER.

Introduction

Much, if not most, of the work in AI planning deals with single problems with full observability, *e.g.*, in classical planning. It is, however, possible to have a plan that solves multiple problems. Depending on the problem's assumptions, the generalized plan can be of different forms.

- If we assume no observability, yet want to achieve the goal in all contingencies, a sequence of actions called a *conformant plan* may work, in which case no matter what the actual world turns out to be, the sequence of actions is executable and leads to a state where the goal condition is satisfied.
- If we have partial observability to differentiate among finitely many cases in the class of planning problems, then a *conditional plan* may suffice, which, based on the observation which of the finite cases the actual world is, takes different actions accordingly.
- If we have infinitely many cases in the problem class, and want to find a solution that achieves the goal in all cases, then an *iterative plan*, *i.e.* a plan with loops or recursion, may be needed.

The last type of planning problems is particularly interesting to us, due to the wide applicability of its solution. For example, the blocks world problem is a well-known task in classical planning. It requires to shuffle a number of blocks from one stack shape into another. This simple problem presents a relatively big challenge to classical, domain-independent planners (Bacchus 2001), in that it usually takes a very long time for them to find a plan to a moderately-sized problem (*e.g.* 50 blocks). However, a compact procedure (loopy plan) exists, which, if one follows the action it proposes at each step, solves any blocks world problem in polynomial time (Gupta and Nau 1992). It would thus be desirable to have an automatic method that is able to find iterative plans like this for any problem class where a procedural solution exists. If this is realized, then classical planning in these domains reduces to the trivial task of instantiating the found loopy plan.

Unfortunately, iterative planning is notoriously difficult. In the most general form, it is equivalent to automatic programming which is not even decidable. Despite its difficulty, many approaches to the problem have been explored and algorithms proposed. To avoid undecidability, some of these approaches sacrifice automation and allow human interaction, others restrict the structural form of loopy plans, and yet others weaken the correctness guarantee.

In the rest of this paper, we first review some of the those approaches, with more details on a planner called KPLANNER (Levesque 2005) that falls into the last category. Based on KPLANNER, we then present a new plan representation and a novel algorithm for generating loopy plans, which lead to FSAPLANNER, a new planner that outperforms its antecedent. Finally, we discuss some of the potential ways to further improve FSAPLANNER's performance, and present some preliminary experimental results.

Related Works

Most early work on iterative planning takes a deductive approach (Biundo 1994), where the plan comes out as a by-product of theorem proving. For example, (Manna and Waldinger 1987) proposes a tableau-based sequent calculus to deductively synthesize recursive plans. They use sequents to represent the dynamics of a planning domain, which consist of assertions, goals and outputs.

Planning proceeds by applying their deduction rules to the sequents, until a "false" assertion or a "true" goal is derived, at which point the associated output in the sequent is the final plan. Recursion is introduced by a well-founded induction rule. However, the induction hypothesis must be provided interactively, and even with human intervention, planning can be very slow.

Magnusson and Doherty recently proposed a deductive planning framework for maintenance goals in temporal action logic (Magnusson and Doherty 2008). In order to get around the invariant identification problem, they incorporated a regularity heuristic and a synchronization heuristic to help the deductive reasoner find useful invariants in such temporal domains. An induction rule is then used to automatically form a plan with loops, after the induction hypotheses as well as the base cases are identified by the heuristics. Magnusson and Doherty showed that their planner solves a practical surveillance problem based on this approach. It remains open whether effective heuristics exists for solving classes of problems with final-state goals.

Deductive approaches like the above generate plans that are provably correct. However, they either cannot be fully automated or work only for a narrow range of problems. Recently, several non-deductive methods are proposed, which relax the correctness guarantee in general, but aim for practicality and automation.

One example is loop-DISTILL (Winner and Veloso 2007), which generates a loopy "domain-specific planner" (dsPlanner) that solves a class of ADL problems. Their algorithm takes as an input a partial order plan to an instance in the problem class. It then identifies the largest matching sub-plan, and converts the repeating occurrences of this sub-plan into a loop. This procedure is repeated greedily, until no more loops can be found. Their algorithm leans heavily on the example plan, and can only generate plans with non-nested loops.

Another method that learns from a single example plan utilizes role-based abstraction (Srivastava, Immerman, and Zilberstein 2007; 2008). In their planner, Aranda, Srivastava *et al.* use state aggregation to group objects of the same role into equivalence classes, and obtain an abstract state representation, where a role is the set of unary predicates that an object satisfies. To obtain a loopy plan, Aranda replaces objects in the example plan with their corresponding abstract objects. At this point, the repeating pattern becomes obvious, and loops can be obtained by folding the repeating sections in the abstract plan. Aranda does not guarantee correctness in general, but they show that it is provably correct in "extended-LL" domains.

Levesque's Approach

Another non-deductive planner for generating loopy plans is KPLANNER designed by Levesque (Levesque 2005). It solves planning problems with a finite number of ground actions and a finite number of primitive fluents, with the dynamics (action preconditions and effects, sensing capabilities, *etc.*) appropriately specified. Fluents are functional and range over finite or countably infinite domains. Among all the fluents, there is a special integer fluent called the *planning parameter*, whose value is unbounded and unknown at planning time. The planning task is to find a plan, possibly with loops, which can solve the planning problem no matter what value the planning parameter actually takes.

For example, in a tree-chopping problem, there may be three fluents: `tree` may be `up` or `down`, `axe` may be `out` or `stored`, and the parameter `chops_needed` is a "secret" integer indicating the number of chops needed to fell the tree.

A `look` action is always possible, and senses whether `tree` is `down` (`chops_needed` has been reduced to 0) or `up` (`chops_needed` not 0 yet). When `axe` is `out`, it is additionally possible to `chop`, which reduces `chops_needed` by 1, and to `store`, which makes `axe` become `stored`.

Initially, `tree` is `up` and `axe` is `out`, and the goal is to make `tree down` and `axe stored`.

The solution to a problem like this is represented by a program-like structure (*e.g.* "dsPlanner" (Winner and Veloso 2007) or "generalized plan" (Srivastava, Immerman, and Zilberstein 2008) in existing work). Levesque formally defined a *robot program* language, whose syntax and meaning is given by Definition 1.

Definition 1 (Robot program (Levesque 1996; 2005))**.** A *robot program* and its execution is defined as

1. **nil** is a robot program executed by doing nothing;
2. for any primitive action A and robot program P, **seq**(A, P) is a robot program executed by first performing A, ignoring any sensing result, and then executing P;
3. for any primitive action with possible sensing result R_1 to R_k, and for any robot programs P_1 to P_k, **case**$(A, [\textbf{if}(R_1, P_1), \cdots, \textbf{if}(R_k, P_k)])$ is a robot program executed by first performing A, and then on obtaining the sensing result R_i, continuing by executing P_i;
4. if P and Q are robot programs, and B is the result of replacing in P some of the occurrences of **nil** by **exit** and the rest by **next**, then **loop**(B, Q) is a robot program executed by repeatedly executing the body B until the execution terminates with **exit** (rather than **next**), and then going on by executing the continuation Q.

According to this definition, intuitively, the following robot program, r_{tc}, is a solution to the tree-chopping problem:

```
loop(
    case(look,
        [   if(down,exit),
            if(up,seq(chop,next))
        ]
    ),
    seq(store,nil)
)
```

The Logical Account for Correctness

In order to formally characterize the semantics of planning problems and robot programs, Levesque uses a logical language called the *situation calculus* (Reiter 2001), with a possible-world extension to handle incomplete knowledge (Scherl and Levesque 2003). Objects in the domain of the logic are of three sorts: *situations*, *actions* and *objects*. S_0 is used to denote the initial situation, and $do(a, s)$ the situation after performing action a in situation s. Functions (relations) whose value may vary from situation to situation are called functional (relational) *fluents*, and denoted by a function (relation) whose last argument is a situation term. The special relation $Poss(a, s)$ states that action a is executable in situation s, $K(s', s)$ means that the agent may think the world is in situation s' when the actual situation is s, and the function $sr(a, s)$ indicates the sensing result of a when performed in s. See (Scherl and Levesque 2003) for how actions, including sensing actions distinguished by sr, affect the K fluent.

The dynamics of a planning problem can be formalized by an action theory Σ consisting of

- facts in the initial situation S_0, including initial beliefs by $K(s_i, S_0)$;
- action precondition axioms of the form $Poss(a, s) \equiv \Pi_a(s)$, one for each action a;
- successor state axioms, one for each fluent f, stating under what condition $f(\vec{x}, do(a, s))$ takes a specific value as a function of what holds in situation s;
- sensed fluent axioms of the form $sr(a, s) = r \equiv \Phi_a(r, s)$, one for each sensing action a, stating under what condition a gets sensing result r in situation s;
- unique names axioms for actions;
- some domain independent foundational axioms on situations;

To formally capture the semantics of robot programs, Levesque defined $Rdo(r, s_1, s_2)$ in the framework of situation calculus, to mean that robot program r, when executed in situation s_1, will legally terminate in situation s_2. The precise definition of $Rdo(r, s_1, s_2)$ can be found in (Levesque 1996), and omitted here due to space reasons.

Let $\phi[s]$ be the goal formula of the planning problem, with a single free variable s, then the planning task for KPLANNER is to find a robot program r, such that

$$\Sigma \models \forall s.\, K(s, S_0) \supset \exists s'.\, \{Rdo(r, s, s') \wedge \phi[s']\}$$

This entailment intuitively means that for any possible initial situation s, the found robot program r will terminate, and result in a state s' in which the goal ϕ is satisfied.

For example, let Σ_{tc} be the action theory of the tree-chopping domain, then the following entailment can be proved.

$$\begin{aligned}\Sigma_{tc} \models \forall s.K(s, S_0) \supset \\ \exists s'.\, \{Rdo(r_{tc}, s, s') \wedge \\ \texttt{tree}(s') = \texttt{down} \wedge \texttt{axe}(s') = \texttt{stored}\}\end{aligned}$$

The planning algorithm

Now the question becomes, "given the planning domain Σ and a goal ϕ, is there an algorithm which can automatically find a robot program r that solves it?" [1]

Considering the difficulty of planning with loops in general, as discussed in the introduction section above, KPLANNER sacrifices the strong notion of correctness for practicality. More specifically, instead of returning a plan that works provably correctly for all planning parameter values, it generates one that is guaranteed to be correct only for two selected integers, namely, a small generation bound N_1 and a larger test bound N_2.

Under this simplification, KPLANNER searches for a plan by alternating between two phases.

- In the *generation phase*, a conditional plan that solves the problem when the planning parameter is known to have value N_1 is generated. This is done by a blind forward search. When such a conditional plan is found, KPLANNER checks if it is a possible unwinding of a robot program with loops, and if so, the loopy plan is forwarded to the test phase.
- In the *test phase*, the robot program obtained from the generation phase is tested with respect to a world where the planning parameter has value N_2. If it works correctly, then it is returned as the solution to the planning problem; otherwise, KPLANNER resumes the generation phase, and a new robot program is enumerated.

Levesque shows that KPLANNER works efficiently in several domains, including tree-chopping, arithmetic, omelette and binary tree searching. Moreover, the resulting robot programs, though only with very weak correctness guarantee, are indeed correct in general.

A New Plan Representation

Levesque claimed that a directed graph representation exists for robot programs, and hypothesized that the two representations are equivalent (Levesque 1996). In this section, we give a formal definition of such a graphical representation called *FSA plan*, and show that it is in fact more general than robot programs.

Definition 2 (FSA Plan)**.** An *FSA plan* is a tuple $\langle Q, \gamma, \delta, q_0, q_F \rangle$, where

- Q is a set of plan states;
- $q_0 \in Q$ is an initial plan state;
- $q_F \in Q$ is a final plan state;

[1] Note that the input to the planning algorithm does not include example solutions. In contrast, recall that both loop-DISTILL and Aranda *learn* from an example plan.

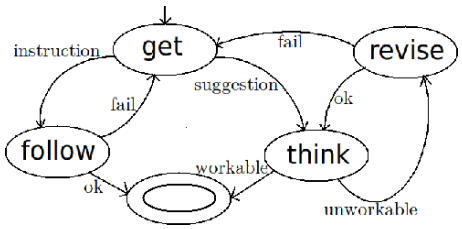

Figure 1: An FSA plan for the find-topic example

- $\gamma : Q^- \to \mathbf{A}$ is a function, where $Q^- = Q \setminus \{q_F\}$ and $\mathbf{A}$ is the set of primitive actions;
- $\delta : Q^- \times \mathbf{R} \to Q \cup \{\perp\}$ is a function, where $\mathbf{R}$ is the set of sensing results, satisfying the constraint that r is a sensing result of $\gamma(q)$ whenever $\delta(q, r) \neq \perp$.

FSA plans can be visualized graphically, where nodes in the graph are plan states in Q, each labelled with its associated action $\gamma(q)$ for $q \in Q^-$. An edge label-ed with r exists between q_1 and q_2 if and only if $\delta(q_1, r) = q_2$. The initial plan state q_0 is denoted by an arrow pointing to it, and the final plan state q_F by a double border.

For example, Figure 1 visualizes an FSA plan for a "find topic" example, where a graduate student wants to find a research topic under the guidance of his supervisor. What he can get from the supervisor is either an instruction or a suggestion. He can follow an instruction: if he succeeds, then he is good; otherwise he has to get some advice again. He can think about a suggestion: if the suggestion is workable, then he is done; if not, he has to revise it. Depending on the revision result, he may have to rethink about the (updated) suggestion, or simply get some other advice from the supervisor.

The execution of an FSA plan $\langle Q, \gamma, \delta, q_0, q_F \rangle$ starts from $q = q_0$, and iteratively does the following:

1. if $q = q_F$, then stop;
2. otherwise, execute action $a = \gamma(q)$;
3. upon getting sensing result r, identify the next plan state q' to follow by $q' = \delta(q, r)$;
4. let $q = q'$ and repeat from Step 1.

Intuitively, an FSA plan is valid for a planning problem if every action it proposes is legal, and the final plan state q_F is always reached, at which point the goal condition of the planning problem is satisfied.

To make this intuition precise, we need to formally define what situation is a final one after executing the FSA plan P from an initial situation s_0. For this purpose, we define a formula $Fdo(P, s_1, s_2)$ (similar to $Rdo(r, s_1, s_2)$ in (Levesque 1996)) to mean that $P = \langle Q, \gamma, \delta, q_0, q_F \rangle$ terminates legally when started in situation s_1, and s_2 is the final situation. Formally, $Fdo(P, s_1, s_2)$ is an abbreviation for the following second-order formula:

$$Fdo(P, s_1, s_2) \stackrel{def}{=} (\forall T). \{ \cdots \supset T(q_0, s_1, q_F, s_2) \}$$

where "$\cdots$" is the conjunction of the universal closure of the following formulae:

1. $T(q, s, q, s)$;
2. $T(q, s, q'', s'') \wedge T(q'', s'', q', s') \supset T(q, s, q', s')$;
3. $\{\gamma(q) = a \wedge Poss(a, s) \wedge SF(a, s) = r \wedge \delta(q, r) = q'\} \supset T\big(q, s, q', do(a, s)\big)$.

From the discussion above, one may find the close resemblance of FSA plan to robot programs. Then the question is: why are we proposing yet another plan representation? In the rest of this paper, we argue that FSA plans are more general than robot programs, and a simple systematic search algorithm exists for finding FSA plans to Levesque's planning with loops problems.

To compare the expressiveness between robot programs and FSA plans, we first need a notion of equivalence.

Definition 3. A robot program r and an an FSA plan P is *equivalent* with respect to an action theory Σ if and only if

$$\Sigma \models \forall s_1, s_2.\ Rdo(r, s_1, s_2) \equiv Fdo(P, s_1, s_2)$$

Based on Definition 3, we have the following theorems.

Theorem 1. For any robot program r, there exists an equivalent FSA plan P.

Proof. By structural induction on robot program constructs. □

This theorem establishes that FSA plans are at least as expressive as robot programs, in that an equivalent FSA plan exists for any robot program. As for whether FSA plans are strictly more expressive, we have:

Theorem 2. There exists an FSA plan that has no equivalent robot program.

Proof. The find-topic example in Figure 1 is such an FSA plan. □

Theorems 1 and 2 lay the theoretical foundation that FSA plans are strictly more general that robot programs given a fixed set of primitive actions, and thus justifies the use of FSA plans as the plan representation in our work. Now we are ready to give a formal definition of the problem of planning with loops in this setting.

Definition 4 (The Planning Problem)**.** Given an action theory Σ and a situation-suppressed goal formula[2] ϕ, the planning task is to find an FSA plan P such that

$$\Sigma \models (\forall s).\, K(s, S_0) \supset \exists s'.\, \{Fdo(P, s, s') \supset \phi[s']\}$$

In other words, the planning problem is to find an FSA plan P such that from any possible initial situation, the execution of P will terminate in a situation where the goal condition is satisfied.

[2]See (Reiter 2001) for a formal definition of situation-suppressed formulae.

```
1:  FsaPlanner(Σ_G, Σ_T, φ, N_1, N_2) {
2:    while(true) {
3:      P = generate(⟨⟩, q_0, P_0)[Σ_G, φ];
4:      if (P == fail)
5:        return fail;
6:      if (test(⟨⟩, q_0, P)[Σ_T, Φ])
7:        return P;
8:    }
9:  }
```

Figure 2: Main program of FsaPlanner

The FSAPLANNER Algorithm

In this section we present a new planner called FsaPlanner for generating FSA plans. Our target problems are similar to those for KPLANNER, where a specification includes a list of primitive actions and their dynamics (preconditions, effects and sensing results), along with a list of primitive fluents and formulae characterizing the initial state and the goal condition.

However, we generalize the idea of planning parameter in KPLANNER: instead of using an integer parameter to range over a class of planning problems, we assume that the initial situation is incompletely specified, and among all the possibilities, two example situations are given to the planner, one for generation and the other for verification. We use Σ_G and Σ_T to represent the action theory for the generation and test problem, respectively. Notice that the treatment with planning parameters in KPLANNER is a special case, since Σ_G here correspond to $\Sigma \cup \{f = N_1\}$ in KPLANNER, and Σ_T to $\Sigma \cup \{f = N_2\}$.

Given Σ_G and Σ_T, the high-level algorithm of FsaPlanner is very similar to that of KPLANNER. Figure 2 is the pseudocode of the main program, where $P_0 = \langle\{q_0, q_F\}, \{\}, \{\}, q_0, q_F\rangle$ is the simplest FSA plan[3] to initiate the search, with one initial state q_0, one final state q_F, and empty γ and δ functions.

The planner switches between a generation phase (lines 3–5) and a test phase (lines 6 and 7). In the generation phase, a new plan P is enumerated, which achieves the goal ϕ with respect to the action theory Σ^G. If no such plan exists, then the planner returns failure; otherwise the planner enters the test phase, where the generated plan P is tested with respect to Σ^T. If P is valid for Σ^T, then it is returned as a solution; otherwise, the planner resumes the generation phase, and a new plan is enumerated.

The main novelty of FsaPlanner lies in the plan generation algorithm. Recall that in KPLANNER, Levesque first generates a conditional plan P' that works for $f = N_1$, and then tries to wind P' into a loopy plan P. A disadvantage of separating searching and winding is that in some cases, even for a compact loopy plan P, the unwinding P' may be a very large conditional plan, and thus takes prohibitively long time to find by progressive search. To overcome this problem, we instead look for a loopy plan directly in the space of FSA plans. Figure 3 shows the pseudocode for the generation phase. Note that due to the existence of nondeterminism in the pseudo code, "**return**" statements should be understood as backtracking points: when a returned FSA plan is rejected in the test phase, then the search in the generation phase resumes from this point, and a new plan is enumerated after backtracking from there.

The intuition behind this algorithm is to maintain a "current" FSA plan, and simulate its execution from the initial state. During the execution, we always try to keep the current plan, and only extend it when necessary. Extensions happen when a transition is missing in δ, in which case we non-deterministically choose a target plan state to change to, and add this transition to δ, introducing a new plan state if necessary.

The procedure *generate* (lines 1–20) revises the current plan $\langle Q, \gamma, \delta, q_0, q_F\rangle$ by simulating its execution from plan state q and history H.[4] It identifies the next action to perform in the current plan state q. If q is the final state, then the goal condition must be satisfied, in which case the current FSA plan is returned (lines 1–7). If q is not final, then it tries to execute the action $\gamma(q)$ associated with q (lines 9–12 and 18), nondeterministically choose an executable one if nothing is associated yet (lines 13–17).

To try an action, the planner must find the correct progression for each of the action's sensing results, and accumulate the transitions, in order to obtain an FSA plan that works for all cases (lines 22–28).

Finally, to progress with respect to an action with a specific sensing result, the planner simply identifies the plan state $q' \in Q$ to transfer to when executing action a in plan state q obtaining sensing result r (lines 30–54). If $\delta(q, a) = q' \neq \perp$ is already in the current FSA plan, then follow this transition, and recursively call the *generate* procedure from there (lines 34–37). Otherwise, non-deterministically choose a $q' \in Q \cup \{q_{new}\}$, add the transition to δ in the current FSA plan, and call *generate* from this q' (lines 38–53), where $q_{new} \notin Q$ is a new plan state. In practice, transfer to existing plan states (lines 40–45) is tried before one to a new state, since compact plans (those with fewer plan states) are preferred. Naturally, if the sensing result is impossible to obtain, then the current plan is returned directly without further search (lines 31–33).

[3] Strictly speaking, $P_\perp = \langle\{q_F\}, \{\}, \{\}, q_F, q_F\rangle$ is the simplest FSA plan, and should be used as the starting FSA in the search. It is a solution to trivial planning problems where the goal is always true in the initial state. However, using P_0 instead of $P_\perp$ makes the presentation simpler, and $P_\perp$ can be treated as a special case.

[4] For our purposes, a history is a sequence of performed actions paired with their sensing results. See (de Giacomo and Levesque 1999) for details.

```
1:  generate(H, q, ⟨Q, γ, δ, q0, qF⟩) {
2:      if (q = qF){
3:          if (Σ^G ∪ {Sensed(H)} ⊨ φ[end(H)])
4:              return ⟨Q, γ, δ, q0, qF⟩;
5:          else
6:              return fail;
7:      }
8:      else {
9:          if γ(q) ≠ ⊥ {
10:             a = γ(q);
11:             γ' = γ;
12:         }
13:         else {
14:             non-deterministically choose a such that
15:                 Σ^G ∪ {Sensed(h)} ⊨ Poss(a, end(H));
16:             γ' = γ ∪ {q → a};
17:         }
18:         return tryAct(H, q, a, ⟨Q, γ', δ, q0, qF⟩));
19:     }
20: }
21:
22: tryAct(H, q, a, P0) {
23:     let r1, ··· , rn be all possible sensing results of a;
24:     for i = 1, ··· , n {
25:         Pi = progress(H, q, a, ri, Pi−1);
26:     }
27:     return Pn;
28: }
29:
30: progress(H, q, a, r, ⟨Q, γ, δ, q0, qF⟩) {
31:     if (r is impossible for a in history H) {
32:         return ⟨Q, γ, δ, q0, qF⟩;
33:     }
34:     else if (δ(q, r) = q' ≠ ⊥) {
35:         return
36:         generate(H · ⟨a, r⟩, q', ⟨Q, γ, δ, q0, qF⟩);
37:     }
38:     else {
39:         non-deterministically
40:         either {
41:             choose q' ∈ Q;
42:             δ' = δ ∪ {⟨q, r⟩ → q'};
43:             return
44:             generate(H · ⟨a, r⟩, q', ⟨Q, γ, δ', q0, qF⟩);
45:         }
46:         or {
47:             choose q' ∉ Q;
48:             Q' = Q ∪ {q};
49:             δ' = δ ∪ {⟨q, r⟩ → q'};
50:             return
51:             generate(H · ⟨a, r⟩, q', ⟨Q', γ, δ', q0, qF⟩);
52:         }
53:     }
54: }
```

Figure 3: The plan generator

Problem	KPLANNER	FSAPLANNER
airport	0.02	0.03
arith	0.8	0.09
bars	0.03	0.24
bintree	0.07	0.1
fact	2.12	0.21
fixedegg (1)	0.0	0.01
fixedegg (2)	0.02	0.01
fixedegg (3)	0.06	0.02
fixedegg (4)	0.31	0.03
fixedegg (5)	7.14	0.06
fixedegg (6)	3051.84	0.09
fixedegg (7)	-	0.15
fixedegg (8)	-	0.21
fixedegg (9)	-	0.32
safe	0.07	0.1
treechop	0.09	0.01
variegg	0.04	0.02
striped+	-	83.26

Figure 4: Comparison between KPLANNER and FSAPLANNER

Experimental Results

We implemented FSAPLANNER in SWI-Prolog based on the abstract algorithm above, and compared the the running time with Levesque's KPLANNER using all of the KPLANNER example problems and the same hand-written search-pruning rules.[5] Like in KPLANNER, we assume the optimal depth of search is unknown, and iterative deepening is used on this parameter.

Figure 4 shows the performance of both planners. In almost all problems, FSAPLANNER behaves better than KPLANNER, sometimes even orders of magnitude faster, *e.g.* in `arith`, `fact`, `fixedegg` and `treechop`. Most notably, KPLANNER was unable to solve `fixedegg` problem when the number of needed eggs is greater than 6. In contrast, FSAPLANNER solves 9 eggs within less than half a second! The last example, `striped+`, is a new and difficult problem inspired by (Srivastava, Immerman, and Zilberstein 2007) (described in detail below). As we can see, KPLANNER is unable to solve this problem even with strong hints, whereas FSAPLANNER solves it (with hand-written pruning rules) within minutes.

Among all the problems, `bintree` and `striped+` require nested loops. This distinguishes FSAPLANNER from loopDISTILL and Aranda, since they do not handle arbitrary loop structure.

In the next experiment, we encoded the three benchmark problems for Aranda (Srivastava, Immerman, and Zilberstein 2008) into our language, and fed them to FSAPLANNER. `delivery` requires to move multiple objects at the dock to their destinations using a truck; `transport` involves delivering pairs of monitors and servers with two types of vehicles; `striped` starts with

[5] www.cs.toronto.edu/~hector/swi-kplanner-1.2.tar.gz

Problem	Aranda		FsaPlanner	
	Time	Plan Size	Time	Plan Size
delivery	14	30	4.84	8
striped	13	33	0.99	9
transport	16	43	-	23

Figure 5: Comparison between Aranda and FsaPlanner

a pile of blocks with equally many blues on top of reds, and the task is to move the blocks via the table, and build a new tower with interleaving colors.

Again, we use the basic algorithm with a blind depth-first search, without any hand-coded pruning rules, but with the optimal depth of search and number of plan states given to the planner. Figure 5 compares the performance of FsaPlanner on the three benchmark problems with that of Aranda.[6]

As we can see, FsaPlanner solves `delivery` and striped very quickly, but `transport` cannot be solved fully automatically. With hand-coded pruning rules, FsaPlanner is able to find a plan with 23 states for `transport`. Although the timing statistics look worse than Aranda in this case, we do not consider it a fundamental flaw of FsaPlanner. For one, Aranda utilizes a state-of-the-art planner to generate its example plans, whereas we use blind forward search at the current stage. It is likely that speedup can be obtained by incorporating a more clever search technique. For another, our implementation is done in Prolog, an interpretive programming language. We expect that a future implementation in C will make our planner more efficient. Notice that in all cases, the optimal depth and size is given to the planner. Without this assumption, a plan can still be found using iterative deepening on these parameters, but it will take much longer time.

In terms of plan size, the loopy plans found by FsaPlanner are more compact than those by Aranda. We believe this is due to FsaPlanner's search policy which expands the FSA plan only when necessary. We also notice that plans found by Aranda are not wound to the greatest extent. If they are further wound by some post-processing, the resulting plans from the two planners may become comparable.

Improvements to the Basic Algorithm

As discussed above, the basic FsaPlanner algorithm in Figure 3 appeals to non-determinism when choosing the next action to execute and the next plan state to transfer to. In practice, this is implemented by depth-first search, which works extremely well when the search space of the planning problem is small (*e.g.* `treechop`, `bintree`, `variegg`), but takes prohibitively long to generate any plan when the search space is large, *e.g.* `transport` and `striped+`, without hand-written pruning rules.

In this section, we discuss two possible improvements to FsaPlanner: adding heuristic action selection and randomizing sensing results in the search.

Heuristics for Action Selection

Heuristic search has brought a huge success to classical planning (Hoffmann and Nebel 2001; Helmert 2006), and can be applied to our basic algorithm when a non-deterministic choice has to be made.

One such non-deterministic choice is the selection of action to associate with a non-final plan state (line 14 in Figure 3). We try a goal-distance based heuristics so that more promising actions are tried before the apparently less fruitful ones.

To estimate the goal distance from a situation, we make the simplistic assumption that all fluents are independent, and each fluent needs to change from its current value to one that *may* satisfy the goal condition. We count the minimum number of action steps needed to change each fluent, and the sum of steps across all fluents is used as the distance estimate. This is similar to the additive heuristics used in HSP (Bonet and Geffner 2001).

One complication is that fluents may have infinitely many possible values. As a result, breadth-first search must be used, instead of the Dijkstra's algorithm for shortest path in finite graphs. This renders the calculation of heuristic value to have exponential complexity in theory. However, as can be seen from Experimental Results, this calculation is pretty fast in practice.

Randomization of Sensing Results

For an action with multiple sensing results, we always analyze them in a predefined order in the basic algorithm (line 24 of Figure 3). This may postpone the discovery of some bad choices that has been made in the search, and require many more backtracking steps before the search returns to the correct branch.

One solution to this problem is to randomize the order of exploration whenever more than one sensing result is possible. In this way, the probability of repeatedly making a same bad choice becomes low, and the bad choice can be detected earlier in the search.

Some Preliminary Result

We implemented both improvement ideas, and compared the performance of four variants of FsaPlanner on the `striped+` example that we designed as a challenging benchmark problem. It is similar to the `striped` benchmark in (Srivastava, Immerman, and Zilberstein 2008), except for two differences. For one, we assume that we have three stacks: A, B and C. Initially, all blocks are on stack A, and the goal is to use B as an intermediate stack, and make C a striped stack. Note that in `striped`, the table serves as the intermediate block holder, and can have infinitely many blocks

[6] Statistics for Aranda are our estimates based on the figures in (Srivastava, Immerman, and Zilberstein 2008).

Problem	FP_M	FP	FP_H	FP_{HR}
striped+	6.23	6179.05	31.8	28.09

Figure 6: Comparing four variants of FsaPlanner

directly on it. Here in `striped+`, however, stack B is the only intermediate holder, so it is much more restricted than `striped` and thus requires more delicacy in the solution plan. For the other, we assume that the color of the blocks in the initial stack is unknown. Each block can be either red or blue. Without the assumption that we have equally many reds and blues, one color may have more blocks than the other, so the goal is to make C a maximal striped tower, and all the leftovers should be placed on B.

We compare the run time for the `striped+` problem with four variants of FsaPlanner.

- FP_M: FsaPlanner with hand-written pruning rule;
- FP: FsaPlanner without pruning rule;
- FP_H: FsaPlanner with goal distance based heuristics;
- FP_{HR}: FsaPlanner with goal distance based heuristics and randomized sensing result.

In all cases, we assume the optimal depth of search and number of plan states are known.

Figure 6 shows the statistics for all four variants on the `striped+` problem. As we can see, planning takes an extremely long time if blind depth-first search (FP) is used, three orders of magnitude slower than the version with elaborated hand-written pruning rules (FP_M). The incorporation of domain-independent, goal-distance based heuristics (FP_H) also speeds up the search dramatically, but still considerably slower than with manual control. Finally, using randomization of sensing outcomes together with heuristic search renders the run-time of the algorithm a random number. In some runs, it takes shorter time (*e.g.* 28.09 *vs.* 31.8), but in others, it may also be extremely long. It is interesting future work to investigate the distribution of runtime, which in turn may give us some insight on how to improve the randomized algorithm.

Conclusion

In this paper, we continue our investigation on iterative planning from a new perspective. The work is closely related to KPLANNER, but our result here is more general in three aspects. First, we formally defined FSA plan, a type of generalized plan provably more general than robot programs on which KPLANNER is based. Second, compared with KPLANNER, we enlarged the scope of problems that can be formulated and solved by loosening the restriction that the planning problems have to be indexed by a planning parameter. Finally, we presented a new algorithm and its implementation FsaPlanner, which is shown to be more efficient than KPLANNER, and highly competitive with other existing iterative planners like Aranda.

Preliminary experiments have shown that FsaPlanner has potential for further improvements. In particular, heuristic search methods can be easily incorporated. It is interesting and promising future work to find an effective domain-independent heuristics that works for a broader class of iterative planning problems.

References

Bacchus, F. 2001. AIPS'00 planning competition. *AI Magazine* 22(3):47–56.

Biundo, S. 1994. Present-day deductive planning. *Current Trends in AI Planning* 1–5.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129:5–33.

de Giacomo, G., and Levesque, H. 1999. Projection using regression and sensors. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 160–165.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2–3):223–254.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

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

Levesque, H. 1996. What is planning in the presence of sensing. In *Proceedings of National Conference on Artificial Intelligence*.

Levesque, H. 2005. Planning with loops. In *Proceedings of International Joint Conference on Artificial Intelligence*.

Magnusson, M., and Doherty, P. 2008. Deductive planning with inductive loops. In *AAAI-08*.

Manna, Z., and Waldinger, R. 1987. How to clear a block: a theory of plans. *Journal of Automated Reasoning* 3(4):343–377.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Scherl, R., and Levesque, H. 2003. Knowledge, action, and the frame problem. *Artificial Intelligence* 144.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2007. Using abstraction for generalized planning. Technical report, Department of computer science, University of Massachusette.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *AAAI-08*.

Winner, E., and Veloso, M. 2007. LoopDISTILL: Learning domain-specific planners from example plans. In *Proceedings of ICAPS-07 Workshop on AI Planning and Learning*.