Planning with Loops

Hector J. Levesque[*]
Dept. of Computer Science
University of Toronto
Toronto, Ont. M5S 3H5
hector@cs.toronto.edu

Abstract

Unlike the case for sequential and conditional planning, much of the work on iterative planning (planning where loops may be needed) leans heavily on theorem-proving. This paper does the following: it proposes a different approach where generating plans is decoupled from verifying them; describes the implementation of an iterative planner based on the situation calculus; presents a few examples illustrating the sorts of plans that can be generated; shows some of the strengths and weaknesses of the approach; and finally sketches the beginnings of a theory, where validation of plans is done offline.

1 Introduction

The vast majority of the work in AI planning today deals with *sequential* planning, generating a sequence of actions to achieve a goal. A smaller community is concerned with *conditional* planning where plans can be tree-like structures, and an even smaller community is concerned with *iterative* planning, where plans can be graph-like structure with loops. The reason for this is clear: sequential planning admits interesting applications, and yet is already quite difficult, even under the assumption of complete knowledge about the initial state.

The bulk of the work on iterative planning is based on theorem-proving (see [Biundo, 1994] for a survey, but see also Section 4 below for some exceptions). From this perspective, plans are viewed as programs, and planning as a kind of program synthesis [Manna and Waldinger, 1980]. This is a notoriously difficult problem, and reasoning about the correctness of programs with loops, *e.g.* in terms of partial correctness and termination, requires mathematical induction and non-trivial algebra. The difficulty, in other words, is not at all like the difficulty with sequential planning where the size of the search space for long plans is the main problem; even short iterative programs can be quite difficult to reason about. Stephan and Biundo [1996] say "Plan generation on this level is an interactive process with non-trivial inferences that in our opinion, which is shared by other authors as well [*citations omitted*], cannot be carried out in a fully automatic way." As far as we can tell, they would say the same today.

So is fully automated iterative planning completely hopeless? Perhaps. But faced with an intractable reasoning problem, we can look for compromises. In this paper, we forego the strong guarantees of correctness provided by the theorem-proving approach. We consider a new way of generating iterative plans that does not traffic in loop invariants, non-negative decreasing expressions, or any of the other items associated with proving programs correct. The resulting plans will come with much weaker guarantees; plan validation will need to be done separately.

The application we have in mind is the sort of high-level programming typical of *cognitive robotics*, *e.g.* [Lespérance *et al.*, 1999]. Here we expect users to provide programs that tell a robot what to do at a very high-level, with considerable nondeterminism left to the robot to deal with at runtime. Part of the nondeterminism can be in the form of declarative goals to achieve. They will require planning on the part of the robot, but the plans are expected to be small relative to the overall mission of the robot. The main contribution of this paper is a new way of generating small plans with loops in this setting.

In the rest of this section, we describe a motivating example, define the class of plans we are searching for, and present the general approach. In Section 2, we discuss the implementation of a system called KPLANNER and the novel way it generates loops. In Section 3, we present three examples of KPLANNER in use. In Section 4, we discuss its limitations and related work. In Section 5, we present the beginnings of a theoretical foundation. In Section 6, we conclude.

1.1 A motivating example

The problem we wish to consider is that of chopping down a tree and putting away the axe [Sardina *et al.*, 2004]. We have at our disposal two primitive actions: *chop*, which hits the tree once with the axe, assuming the tree is up and the axe is available, and *store*, which puts the axe away, assuming it is available. We observe the following:

1. With no additional information, the problem is insoluble. There is no way to know when or if the tree will go down, and when to put away the axe.

[*]This research was done while I was away on sabbatical at York University in Toronto, RWTH in Aachen, Universita di Roma La Sapienza, the University of New South Wales in Sydney, and Simon Fraser University in Vancouver. I am very grateful to my hosts at these institutions for providing me with a wonderful environment for not doing teaching and administration. Sabbaticals rock!

2. If we are told that the tree will go down if it is hit 3 times, the problem is solved with the following sequential plan:

```
chop ; chop ; chop ; store
```

3. If we are told that the tree will go down if it is hit *at most* 2 times, the problem is again insoluble. But if we are given a third action, *look*, which is a *sensing action* [Reiter, 2001] telling us whether the tree is down or up, then the problem can be solved with the following conditional plan:

```
CASE look OF
   -down: store
   -up: chop ;
        CASE look OF
           -down: store
           -up: chop ; store
        ENDC
ENDC
```

4. If all we are told is that the tree will *eventually* go down if it is hit repeatedly, and we have the *look* action, the problem can be solved with the following iterative plan:

```
LOOP
  CASE look OF
     -down: EXIT
     -up: chop ;
          NEXT
  ENDC
ENDL ;
store
```

Let us call this plan *TC* (TreeChop) for reference.

So in the most general case where the goal is achievable, an iterative plan like *TC* is necessary to achieve it. Moreover, *TC* is *general*, in that it handles all the cases where a conditional plan would be sufficient too.

1.2 Robot programs

Once we move beyond simple sequential plans, it becomes necessary to specify exactly what we mean by a plan. It is convenient here to use a variant of the *robot programs* of [Levesque, 1996] for this purpose. Assume we are given a set of primitive actions, some of which may be sensing actions each having a finite set of possible sensing results. Robot programs and their execution are defined by the following:

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 result, and then executing P;
3. for any primitive action A with possible sensing results R_1 to R_k, and for any robot programs P_1 to P_k, **case**$(A, [$**if**$(R_1, P_1), \ldots,$ **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.

A formal definition of the execution of robot programs in the situation calculus is given in [Levesque, 1996]. The example plans shown earlier, including *TC*, are pretty-printed versions of robot programs. Although quite limited in form (no recursion, no logical expressions, no fluents, no variables), there is a precise sense in which robot programs (with the inclusion of five special primitive actions) are *universal* [Lin and Levesque, 1998]. So for us, planning is this: given a goal, find a robot program that achieves it.

1.3 The planning approach

We are going to try to solve planning problems like the tree chopping where some unknown quantity must be dealt with. Specifically, we assume that among the fluents we are dealing with, there will be a single distinguished one, which we will call the *planning parameter*, that has the following properties: (1) its value is not known or even bounded at plan time, and (2) no loops would be required to achieve the goal if its value were known. In other words, loops are required (if at all) only to deal with an unknown and unbounded planning parameter. For tree chopping, the planning parameter is the number of chops needed to fell the tree.

Given an application domain with a planning parameter F, rather than generating a plan and proving that it works for *all* values of F (*e.g.* by using partial correctness and termination), we will work with two bounds on F and try to find plans with loops using these bounds. In particular, we assume the following:

- the user will provide a constant N_1 (called the *generating bound*) and we *generate* a plan (possibly containing some number of loops) that is provably correct under the assumption that $F \le N_1$;
- the user will provide a second larger constant N_2 (called the *testing bound*) and we *test* that the plan generated in the first step is also provably correct under the assumption that $F \le N_2$.

So the only guarantee we get out of this is that the plan we return will be correct assuming that $F \le N_2$. This is far from foolproof. For example, the conditional plan for tree chopping presented above works correctly for $N_1 = 1$ and $N_2 = 2$, but fails when $F \ge 3$. Suppose, however, that the user had specified $N_1 = 1$ and $N_2 = 100$. Then the smallest conditional plan that satisfies both bounds would have $3 \times 100 + 1$ actions in it. If we generate plans using N_1 only, and *we consider smaller plans before larger ones*, we will generate desired iterative plans well before we encounter undesired conditional ones (and see the theory in Section 5).

2 The planner

So the planning setup, then, is the following:

- the user provides a *problem specification* defined by a list of primitive actions and fluents, and formulas characterizing the initial and goal states, and for each action, its preconditions, effects, and sensing results;
- the user also identifies the parameter F and supplies the two numbers N_1 and N_2;

- we generate a plan that is correct for $F \leq N_1$; because N_1 is small, this search can be done reasonably efficiently, as we will see;
- we test the plan to see if it is correct for $F \leq N_2$; because this involves only testing a given plan, this can also be done reasonably efficiently, even for large N_2.

In practice, and on the examples considered to date, even for very small N_1 and for very large N_2, almost all of the time is spent finding a plan that works for $F \leq N_1$.

The planner, called KPLANNER, is written in Prolog, and consists of three main modules: a plan tester, a plan generator, and a formula evaluator used by the other two modules.[1]

We will describe the formula evaluator in more detail when we look at some examples below. For now, it suffices to note that we need to be able to handle both the *projection* task (determining if a formula is true after a sequence of actions) and the *legality* task (determining if an action can be executed). We use regression for both [Reiter, 2001]. Moreover, we need to be able to incorporate the putative results of sensing, and use them in the evaluation. So instead of just keeping track of a sequence of actions performed to date, we maintain a *history* consisting of pairs of actions and sensing results. (For actions without sensing results, we use *ok* as the result.) Finally, we need to be able to determine when a sensing result cannot occur based on the history to date. For example, for tree chopping where we know that $F \leq 2$, the *look* action cannot produce a sensing result of *up* in a history that contains two legal chop actions.

2.1 Testing plans

Given a plan P and a testing bound N, we need to determine if P correctly achieves the goal assuming that $F \leq N$. Since we have a bound on F, we do not need to prove a theorem, but can simulate the execution of P for all possible results of the sensing actions and confirm that the goal is satisfied in all cases. We have the following: A plan P achieves a goal G starting in history H (initially empty) if

1. $P = \textbf{nil}$ and G holds in history H;
2. $P = \textbf{seq}(A, P')$, the precondition of A holds in H, and P' achieves G in history $H \cdot (A, ok)$;
3. $P = \textbf{case}(A, L)$, the precondition of A holds in H, and for each $\textbf{if}(R_i, P_i) \in L$ such that R_i is a possible sensing result for A given that $F \leq N$, P_i achieves G in history $H \cdot (A, R_i)$;
4. P is a loop that unwinds to Q, and Q achieves G in history H.

The unwinding of $\textbf{loop}(B, C)$ is B but with all occurrences of **exit** replaced by C and with all occurrences of **next** replaced by $\textbf{loop}(B, C)$ itself.[2] Observe that unwinding a loop produces a plan that executes the same way as the original loop. This will be significant when it comes to generating plans.

[1] The code for KPLANNER and a number of examples can be found at `http://www.cs.toronto.edu/cogrobo`.

[2] To guard against loops that run forever, such as $\textbf{loop}(\textbf{next}, \textbf{nil})$, we also need an upper bound on the total number of unwindings that will be performed.

2.2 Generating plans

Given a generating bound N, we use a variant of *progressive planning* to produce a plan that correctly achieves the goal assuming that $F \leq N$. We have the following: To find a plan that achieves a goal G starting in history H (initially empty),

1. if G holds in H, return **nil**;
2. otherwise, select a primitive action A whose precondition is satisfied in H (and maybe other criteria too);
3. if A has no sensing results, do the following:
   (a) find a plan P that achieves G starting in $H \cdot (A, ok)$;
   (b) return the plan $\textbf{seq}(A, P)$;

   if A has sensing results, then do the following:
   (a) for each sensing result R_i that is possible in H given that $F \leq N$, find a plan P_i that achieves G starting in history $H \cdot (A, R_i)$;
   (b) return the plan $\textbf{case}(A, L)$, where L is the list of $\textbf{if}(R_i, P_i)$ for the R_i and P_i from the previous step (the remaining sensing results R_i are impossible, and so the corresponding P_i are "don't care");
4. if the plan in the previous step is the unwinding of a loop, return the loop as well (as described below).

2.3 Generating loops

The key question is this: Where are plans with loops going to come from, if not from the proof of a general, universal theorem? As we suggested in the generation procedure above, they can come from sequential and conditional plans that have already been generated.

To see how this works, consider two conditional plans that are correct for tree chopping given that $F \leq 1$:

```
CASE look OF         CASE look OF
  -down: store         -down: store
  -up: chop ;          -up: chop ;
       store                CASE look OF
ENDC                          -down: store
                              -up: don't care
                            ENDC
                     ENDC
```

The one on the right does a sensing action that is not needed since that *up* sensing result is impossible for $F \leq 1$. Moreover, that plan remains correct for $F \leq 1$ for any substitution of the "don't care." The key observation here is that there is a substitution for which that plan becomes the unwinding of a loop: if we replace the "don't care" by the robot program $\textbf{seq}(\textit{chop}, \textit{TC})$ where *TC* is the plan above, then the program that obtains is an unwinding of *TC*.[3] The conclusion: *TC* is also correct for $F \leq 1$ and can be returned as a potential plan to be tested for the larger bound (in this case, successfully).

But for this idea to be practical, it must be possible to quickly check if a plan matches the unwinding of a loop (or the unwinding of an unwinding *etc.*). How can we do this without just guessing at the loop? Here is where using Prolog as our implementation language pays off: we can write an `unwind` predicate, so that `unwind`(P, Q) holds if loop

[3] More precisely, it is the unwinding of an unwinding of *TC*.

```
  %  P is a loop that unwinds to Q.
unw(P,Q) :- P=loop(B,C), sub(B,P,C,Q).

  %  sub(B,X,Y,Q) holds when Q is the result
  %  of replacing in B each 'exit' by Y and
  %  each 'next' by an unwinding of X
sub(_,_,_,Q) :- var(Q), !.
sub(exit,X,Y,Q) :-not X=loop(exit,_), Q=Y.
sub(next,X,_,Q) :-not X=loop(next,_), unw(X,Q).
sub(seq(A,P),X,Y,seq(A,Q)) :-sub(P,X,Y,Q).
sub(case(A,U),X,Y,case(A,V)) :-subl(U,X,Y,V).
sub(loop(G,P),X,Y,loop(G,Q)) :-sub(P,X,Y,Q).

subl([],_,_,[]).
subl([if(R,P)|U],X,Y,[if(R,Q)|V]) :-
        sub(P,X,Y,Q), subl(U,X,Y,V).
```

Figure 1: Prolog code for an unwind predicate

P unwinds to Q, but then call it by passing it a Q and having it return a P. Simplified code for doing this is in Figure 1. It is easy to confirm that when the Q is the conditional plan on the right above (with a Prolog variable as the "don't care"), the first P it returns is the desired iterative plan *TC*.

This method of generating loops has turned out to be significant. The alternative of enumerating all possible plans containing loops is not practical even for very small plans.

3 The planner in action

We are now ready to consider KPLANNER in action on some problem specifications provided by the user. The representation we use is a variant of the one in INDIGOLOG [de Giacomo and Levesque, 1999; Sardina *et al.*, 2004] based on the situation calculus [McCarthy and Hayes, 1981; Reiter, 2001]. The user supplies the definition of nine predicates in Prolog, which we describe below.[4]

As in INDIGOLOG, all fluents in KPLANNER are functional. Unlike INDIGOLOG, fluents are interpreted *epistemically*, in that we take them to have more than one possible value, according to what is currently known [Vassos, 2004]. This allows us to reason about sensing without some of the disadvantages of possible worlds [Demolombe and Parra, 2000].

The predicates described below take as their arguments *conditions* which are logical formulas, closed under boolean operators and quantifiers. The atomic formulas are arbitrary Prolog goals, except that they may contain fluents. These are evaluated by replacing the fluents by their values and then calling Prolog on the result. We say that a formula is *possibly true* if the goal succeeds for some possible value of the fluents; the formula is *known to be true* if the goal succeeds for every possible value of the fluents.

The Prolog predicates defined by the user are (for action a, fluent f, sensing result r, condition c and arbitrary value v):

- `prim_fluent`(f), declares f as fluent;
- `prim_action`$(a, [r_1, \ldots, r_n])$, declares a to have the r_i as possible sensing results; when $n = 1$, the action is considered to provide no sensing information;

[4]There are also directives to help control the search, which we do not describe further here.

```
prim_fluent(axe).
prim_fluent(tree).
prim_fluent(chops_max).

prim_action(chop,[ok]).
prim_action(look,[down,up]).
prim_action(store,[ok]).

poss(chop,and(axe=out,tree=up)).
poss(look,true).
poss(store,axe=out).

init(axe,out).
init(tree,up).
init(tree,down).

causes(store,axe,stored,true).
causes(chop,tree,down,true).
causes(chop,tree,up,true).
causes(chop,chops_max,X,X is chops_max-1).

settles(look,X,tree,X,true).
rejects(look,up,chops_max,0,true).
settles(look,down,chops_max,0,true).

parm_fluent(chops_max).
init_parm(generate,chops_max,1).
init_parm(test,chops_max,100).
```

Figure 2: A problem specification for tree chopping

- `poss`(a, c), c is the precondition for a;
- `init`(f, v), v is a possible initial value for f;
- `causes`(a, r, f, v, c), when c holds, a causes f to get value v; more precisely, any v for which c is possible becomes a possible value of f; (the r is optional)
- `settles`(a, r, f, v, c), when c is known, after doing a and getting result r, f is known to have value v;
- `rejects`(a, r, f, v, c), when c is known, after doing a and getting result r, f is known not to have value v.

The final two predicates are not part of the action theory, but are used to specify the planning parameter, and its possible values for generating and for testing:

- `parm_fluent`(f), fluent f is the planning parameter;
- `init_parm`(w, f, v), where w is `generate` or `test`, v is a possible initial value for the planning parameter f.

3.1 The tree chopping example

In its most direct formulation, we would formalize tree chopping using a fluent *chops_needed* as the planning parameter. But then to handle a testing bound of 100 would require us to deal with 100 possible values for this fluent. Instead, it is sufficient to keep track of the maximum of these possible values, which we call *chops_max*.

The full problem specification in this language for the tree chopping example is in Figure 2. Since it is not known whether the tree is up or down initially, there are two possible initial values for the *tree* fluent, and similarly after a *chop* action. (Actions like this are sometimes called *nondeterministic*

```
prim_fluent(acc(N)) :- N=1 ; N=2.
prim_fluent(input).   % the unknown fluent

prim_action(incr_acc(N),[ok]) :- N=1 ; N=2.
prim_action(test_acc(1),[same,diff]).

poss(incr_acc(_),true).
poss(test_acc(1),true).

causes(incr_acc(N),acc(N),V,V is acc(N)+1).
settles(test_acc(1),same,input,V,V=acc(1)).
rejects(test_acc(1),diff,input,V,V=acc(1)).

init(acc(_),0).
parm_fluent(input).
init_parm(generate,input,V) :- V=1 ; V=2.
init_parm(test,input,V) :- V=1 ; V=2; V=3.
```

Figure 3: A counting example

in the literature.) The *look* action is what settles its value. In addition, if *look* reports *up*, then *chops_max* cannot be 0. So if we know that *chops_max* is 0 (as a result of having performed some *chop* actions), the *up* result is impossible. We will show the output of a run of KPLANNER on the next example. For this one, suffice it to say that it finds *TC* in .11 seconds.[5]

3.2 A counting example

We turn now to a very different example involving some simple counting. The problem is this: We have two accumulators and some unknown integer input k, where $k > 0$. The primitive actions are: *incr_acc*(n), increment accumulator n (both start at 0); and *test_acc*(1), sense if the first accumulator has the same value as the input. The goal is to make the second accumulator have the value $2k - 1$.[6]

The complete problem specification is in Figure 3. There are three fluents, *acc*(1), *acc*(2), and *input*, the last of which is the planning parameter. A run of KPLANNER with the output it produces is in Figure 4. KPLANNER works by iterative deepening, and there are numbers in the output to indicate the level. An 'x' indicates a generated plan that was sufficient for the generating bound, but that did not work for the testing bound (many of which were omitted from the figure). Note the multiplication in the goal. Nothing in the specification said anything about multiplication. Although progressive planning has serious disadvantages, one advantage it does have is that all we need to be able to do is test if a goal condition is satisfied; we do not need to reason about the goal in a more analytic way.

3.3 A searching example

We now turn to a much more complex example, that of searching an unbounded binary tree for a target node.[7] More precisely the primitive actions are (1) *check_node_type*: sense if the current node is a target node, a non-target leaf node, or a non-target internal node (having left and right children); (2) *push_down_to*(x): go down from an internal node to the left or right, x; and (3) *pop_up_from*: return from a child node. A few moments thought should convince the reader that this problem cannot be solved without additional storage, and so we assume that *push_down_to*(x) pushes the direction x onto an internal *stack*, and that *pop_up_from* pops the stack and produces the popped value as a sensing result.

The rest of the specification is too large to display here. We use a fluent *depth_max* much like *chops_max* so that we cannot get a node type of *internal* when *depth_max* is 0.

But when do we know that we cannot get a node type of *leaf*? This is a bit trickier. The answer is: when there are no more nodes left to explore! For example, assuming we search left branches before right ones, then when we are on the rightmost branch of a tree, we can only get node types of *internal* or *target*. This is reflected in the following:

```
rejects(check_node_type,leaf,stack,S,
     all(x,member(x,S),last_dir(x))).
```

Getting a result of *leaf* rejects any stack of moves whose members are all for the last direction available. Without this constraint, the problem is insoluble; with it, we get the very nice plan shown in Figure 5. Note that KPLANNER generates a nested loop; we believe that this nesting is required here.

```
The goal:  acc(2) is 2 * input - 1
0 1 2 3 4 5xx ... 6xx ... 7xx ...
A plan is found after 1.42 seconds.
-------------------------------------------
incr_acc(1) ;
LOOP
  CASE test_acc(1) OF
     -same: EXIT
     -diff:
        incr_acc(1) ;
        incr_acc(2) ;
        incr_acc(2) ;
        NEXT
  ENDC
ENDL ;
incr_acc(2)
```

Figure 4: Doing the arithmetic

4 Discussion and related work

Because of the loops, the three examples presented here are clearly beyond the scope of existing sequential and conditional planners. As far as we know, no planner based on fully automated theorem-proving can generate the three plans either. Two other camps not based on theorem-proving have considered an interesting special case of iterative planning, for what might be called *repeated attempt* problems.

First, with probabilities. Consider an action a that has a non-zero probability of making some G true. If we assume repetitions of a have independent outcomes, then an iterative plan like *TC* will achieve G with probability 1. This is just right for trying to pick up a block [Ngo and Haddawy, 1995] or to find a good egg [Bonet and Geffner, 2001]. However, independence is untenable for tree chopping, and it is not

[5]All runs were done in Eclipse Prolog version 5.7 on a Mac G5 single 1.6GHz processor with 512MB memory.

[6]This is an abstract variant of the problem of building two copies of a stack of coloured blocks.

[7]This can be thought of as a simplified version of searching for a file in a Unix directory of arbitrary depth.

```
The goal:  current = target
0 1 2 3 4 5 6xxxxx
A plan is found after 0.07 seconds.
-------------------------------------------
LOOP
  CASE check_node_type OF
     -target: EXIT
     -leaf:
        LOOP
          CASE pop_up_from OF
             -left: EXIT
             -right: NEXT
          ENDC
        ENDL ;
        push_down_to(right) ;
        NEXT
     -internal:
        push_down_to(left) ;
        NEXT
  ENDC
ENDL
```

Figure 5: Searching a binary tree

clear what to replace it with. It is even less clear how to exploit probabilities in the counting or search examples above.

A related approach is the model-checking of [Cimatti *et al.*, 2003], where planning problems are cast as finite state systems. For tree chopping, if we are willing to ignore the planning parameter, this works out perfectly: they would get four states and generate (the equivalent of) *TC* almost instantly. However, without that information about the planning parameter, they are forced to conclude that there is no "strong" solution to the problem, only a "strong cyclic" one. Indeed they can never generate a strong solution when loops are required. The counting and search examples above, which depend on the planning parameter in a more direct way, would thus appear to be outside their scope as well.

KPLANNER has its own limitations, however. In particular, it does not scale at all well as the search space grows, even for seemingly easy problems. Consider the problem of getting some good eggs into a bowl [Bonet and Geffner, 2001]. For just one egg, we need a plan with a single loop like in Figure 6. To get three good eggs, we would need a plan consisting of three copies of this plan, strung together sequentially. Without an additional sensing action to tells us if we have enough good eggs, there is not a more compact solution. So with this we can force KPLANNER to generate long plans.

And how well does it do? Here are some timing results:

Number of good eggs	Size of plan (unwound)	Number of backtracks	Running time in seconds
1	6	1	.004
2	9	6	.02
3	12	42	1.41
4	15	702	1681.
5	18	??????	$>$ 3 weeks

As is very clear, unless the search space can be limited in some way, *e.g.* by forward filtering [Bacchus and Kabanza, 2000], KPLANNER is practical only for small plans, in keeping with the cognitive robotics application sketched above. We might say that it is better suited for small but difficult problems than for large but easy ones. For example, it quickly finds a nice solution to the more general problem of getting an arbitrary number of good eggs in the bowl, where a second sensing action now determines when there are enough.

```
LOOP
  break_next_egg_into_dish ;
  CASE sniff_dish OF
     -good_egg: EXIT
     -bad_egg:
         discard_dish_contents ;
         NEXT
  ENDC
ENDL ;
transfer_dish_contents_to_bowl
```

Figure 6: Getting one good egg into a bowl

5 Towards a theory

KPLANNER does a good job of synthesizing small plans with loops, but without a strong guarantee of correctness. While our approach has been to construct a plan in a way that does not require simultaneously proving its correctness, it would of course be very useful to know that the plans were nonetheless correct. Are there conditions under which a plan that works for values of the planning parameter F up to the testing bound will be guaranteed to work for *all* values of F? Without looking too hard for a free lunch, we can sketch a promising direction for further research.[8]

Let $W(k, \sigma)$ stand for the proposition that if we start in any initial state where $F = k$, and we perform the action sequence σ, we end up in a goal state. Let us call a planning problem *simple* wrt action A if it satisfies the following:

1. Sensing can only tell us whether or not the initial value of F was equal to the number of A actions done so far.
2. If an action sequence is legal starting in one initial state but not in another, then at some point in the sequence, the sensing results would be different.
3. If $W(k,\, \alpha\cdot\beta)$ and $W(k+m,\, \alpha\cdot\sigma\cdot\beta)$ and $W(k',\, \alpha\cdot\gamma)$, where $k' > k$, then $W(k'+m,\, \alpha\cdot\sigma\cdot\gamma)$.

The last condition says that if it was sufficient to add σ in going from k to $k+m$, then that σ can also be used for any larger k'. This ensures that loops depend only on the F.

Both the tree chopping and the counting examples can be shown to be simple (wrt *chop* and *incr_acc*(1) respectively). The search example, on the other hand, is nowhere near simple. We get the following theorem:

Theorem 1 *Suppose we have a planning problem that is simple with respect to action A, and a robot program P that contains N occurrences of A. If P is a correct plan for all values of $F \leq N+2$, then P is correct for all values of F.*

This theorem is proved by adapting ideas from the Pumping Lemma of classical automata theory. It suggests a variant

[8]This is joint work with Patrick Dymond and independently with Giuseppe de Giacomo.

of KPLANNER: instead of having the user specify a testing bound, we *compute* a testing bound once a plan P has been generated by counting the occurrences of A in P. Then any plan that passes the test is guaranteed by the theorem to be correct. This applies to simple planning problems only; but we believe that theorems like the above can be found for less restrictive classes of problems. We should not expect to be able to compute testing bounds in this way for *all* planning problems however, since armed with suitable primitive actions, robot programs have the power of Turing machines.

6 Conclusion

We have presented a new way of generating a plan with loops that is not tied to proving a theorem about its correctness. The method involves generating a plan that is correct for a given bound, determining if the plan is the unwinding of a plan with loops, and testing if another unwinding of the plan with loops would also be correct for a larger bound.

Other than the handling of loops, these steps reduce to traditional non-iterative planning, and would benefit from being performed by a more efficient conditional planner, such as the one by Petrick and Bacchus [2004]. Similarly, the winding and unwinding of loops is performed by KPLANNER in a fairly obvious way. Anything that would reduce the number of legal plans considered would speed things up considerably, since for iterative deepening, *all* legal plans of one size must be considered before going on to the next.

On the more theoretical side, we presented a theorem showing that this method of planning is correct for a certain class of simple planning problems. It remains to be seen whether the theorem can be strengthened to include more complex planning problems for which the planner does appear to work, such as the search example presented here.

Acknowledgements

Thanks to Gerhard Lakemeyer and Fahiem Bacchus for helpful comments on an earlier version of the text.

References

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116, 2000.

[Biundo, 1994] S. Biundo. Present-day deductive planning. In C. Bäckström and E. Sandewell, editors, *Procs. of the 2nd European Workshop on Planning (EWSP-93)*, pages 1–5, Vadstena, Sweeden, 1994. IOS Press (Amsterdam).

[Bonet and Geffner, 2001] B. Bonet and H. Geffner. Gpt: a tool for planning with uncertainty and partial information. In *Proc. IJCAI-01 Workshop on Planning with Uncertainty and Partial Information*, pages 82–87, 2001.

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

[de Giacomo and Levesque, 1999] G. de Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.

[Demolombe and Parra, 2000] R. Demolombe and M. Pozos Parra. A simple and tractable extension of situation calculus to epistemic logic. *Lecture Notes in Computer Science*, 1932, 2000.

[Lespérance *et al.*, 1999] Y. Lespérance, H. Levesque, and R. Reiter. A situation calculus approach to modeling and programming agents. In A. Rao and M. Wooldridge, editors, *Foundations and Theories of Rational Agency*. Kluwer, 1999.

[Levesque, 1996] H. Levesque. What is planning in the presence of sensing? In *Procs. of the 13th National Conference, AAAI-96*, pages 1139–1146, Portland, Oregon, 1996.

[Lin and Levesque, 1998] F. Lin and H. Levesque. What robots can do: Robot programs and effective achievability. *Artificial Intelligence*, 101:201–226, 1998.

[Manna and Waldinger, 1980] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[McCarthy and Hayes, 1981] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Webber and N. Nilsson, editors, *Readings in Artificial Intelligence*, pages 431–450. Kaufmann, Los Altos, 1981.

[Ngo and Haddawy, 1995] L. Ngo and P. Haddawy. Representing iterative loops for decision theoretic planning. In *Extending Theories of Action: Papers from the 1995 AAAI Spring Symposium*, pages 151–156. AAAI Press, Menlo Park, 1995.

[Petrick and Bacchus, 2004] R. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11, Menlo Park, CA, June 2004. AAAI Press.

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

[Sardina *et al.*, 2004] S. Sardina, G. de Giacomo, Y. Lespérance, and H. Levesque. On the semantics of deliberation in IndiGolog – from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299, Aug 2004.

[Stephan and Biundo, 1996] W. Stephan and S. Biundo. Deduction based refinement planning. In B. Drabble, editor, *Procs. of AIPS-96*, pages 213–220. AAAI Press, 1996.

[Vassos, 2004] S. Vassos. A feasible approach to disjunctive knowledge in situation calculus. Master's thesis, Department of Computer Science, University of Toronto, 2004.