Iterative Planning: A Survey

Yuxiao (Toby) Hu

October 23, 2008

Abstract

This paper surveys existing work on iterative planning.[1] It starts with the motivation and background knowledge in Section 1, and then reviews some existing approaches to finding loopy plans in Section 2. Section 3 discusses about the underlying theory, which is followed by the possible directions for future work in Section 4 and conclusion in Section 5.

1 Introduction

In this section, I will introduce classical planning and the motivation for generating program-like plans. A logic for reasoning about such planning domains, along with candidate languages for representing the plans, will then be introduced. Finally, some related problems, including program synthesis and grammar induction, will be mentioned, and their differences from planning with loops discussed.

1.1 Automated Planning

In artificial intelligence, the area of *automated planning* usually refers to finding a sequence of legal actions whose execution will change the world from the specified initial state to one that satisfies some goal condition [32].

A classical model for planning is STRIPS [7], where the problem is described by *a set of conditions* that characterize what is true in a state, *a set of action operators* that transforms a state into another, *an initial state*, and *a goal condition*. States and conditions are typically represented by a list of atomic formulas that hold in them. An action operator consists of a *precondition* that must be satisfied before the the action can be executed, an *add list* of facts that will become true after its execution and *a delete* list of facts that will cease to be true. A plan for a STRIPS problem is a sequence of ground action, which is guaranteed to be executable from the initial state, and the goal condition is satisfied after the execution.

[1] I use the terms *iterative plan* and *loopy plan* interchangeably to refer to program-like plans possibly with loops and/or recursions.

The STRIPS model of planning is extended by Pednault in his ADL to handle disjunctive and quantified conditions, conditional effects, *etc* [27]. With the introduction of the International Planning Competition [8] in 1998, McDermott *et al.* proposed PDDL [11], a standard language for describing planning domains and problems that includes STRIPS and ADL as fragments. Recently, the planning community has explored further extensions including numerical fluents, time, concurrency [9], temporally-extended constraints and preferences [10],*etc.*

All the work in this branch shares one thing in common: the solution to the planning problem is a sequence of actions, due to the determinism of actions and states. We call this type of problem *sequential planning.*

Consider a STRIPS problem in the blocks world, where block A is initially on C, C on B, and B on the table. If the goal is to move all the blocks to the table, then the following action sequence is a valid plan:

unstack(A,C); putdown(A); unstack(C,B); putdown(C)

where $unstack(X, Y)$ means to pick up block X which is located on Y, and $putdown(X)$ means to put X onto the table.

As another example, consider a "tree-chopping" problem,[2] where the goal is to let down a tree and put away the axe. Assume that two consecutive chops will bring the tree down, provided the tree is up before the chops and the axe is available, and a store action will put away the axe. If in the initial state, the tree is up and axe out, then the sequential plan

```
chop; chop; store
```

is a valid solution to the planning problem.

Another type of planning where the plan is a straight line sequence of actions is *conformant planning* [35], where both the states and the actions may be non-deterministic. The task is to find a linear plan, whose execution guarantees to achieve the goal, no matter what the true state or effect of actions turn out to be within the non-determinism. In conformant planning, the world state is assumed to be unobservable, *i.e.*, there is no way for either the planner or the plan executor to rule out any possible world within the known non-determinism.

For instance, in the tree chopping example above, if we only know that the tree may need 0, 1 or 2 chops to fell, the problem is insoluble, because it is not known whether the tree is up (needing 1 or 2 chops) or down (needing 0 chop), and chopping is only possible if the tree is known to be up. However, if we assume chopping is allowed as well even when the tree is down, then the plan

```
chop; chop; store
```

will still work, although the chop actions may or may not be necessary.

If one assumes that the world is observable but possibly non-deterministic, then the plan need no longer be a sequence, but instead, may have branches based on the facts at run time in the real world. This type of problem is called *conditional planning.*

[2]Both the problem and the plan representation are adapted from [16].

There are two different ways to formalize how a plan can gain this run-time information. The first is based on the idea of *sensing actions*. For example, in the tree-chopping problem, if we return to the assumption that chop is only possible when the tree is up, but assume there is a sensing action "look", which will tell the plan executor whether the tree is up or down, then the following conditional plan will solve the problem:

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

The second way is to assume the plan has access to some observation variables in the domain at run time, and can branch according to their values. In the tree chopping example, if we assume there is no sensing action, but the plan executer knows the condition of the tree, then we may have a different style of conditional plan:

```
IF tree=down THEN store
             ELSE chop
                  IF tree=down THEN store
                               ELSE chop
                                    store
                  ENDIF
ENDIF
```

Existing work on conditional planning includes Petrick and Bacchus [28, 29], Bertoli *et al* [1], *etc.*

1.2 Motivation for Iterative Planning

The planning tasks above are concerned with individual problems, *e.g.*, the blocks problem only asks for a plan for three blocks on one stack, and the tree chopping only for a plan where (at most) two chops are needed.

If we are given a new problem with 5 blocks in two stacks or a tree that needs 8 chops, for example, then the planner has to be run from scratch to find the plans, though these instances seem to be in the same "style" as the previous cases.

It would thus be ideal to find a plan that works for a *class* of problems, so that it can achieve the goal for any problem instance in the class. For example, in the blocks world, we may like to have a plan, which can move all the blocks onto the table, no matter how many blocks there are and how they are stacked; in the tree chopping example, it would be desirable to have a plan that chops the tree down no matter how many (finite) chops are actually needed to fell it.

For this purpose, sequential and conditional plans are not sufficient, and we need loops (or recursion) in the plan [16].

For example, if we are only told that the tree will *eventually* fall, but the required number of chops is unknown, the following plan with a loop will do.

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

This loopy plan can be considered a generalization of the conditional plan in the previous subsection, in that it works for any tree-chopping problem with arbitrary (finite) number of required chops.

Similarly, in the blocks world, the following loopy plan always achieves the goal to move all the blocks to the table.[3]

```
LOOP
    CASE exist_clear(X)_on(Y) OF
        - yes: unstack(X,Y);
               putdown(X)
        - no: EXIT
    ENDC
ENDL
```

Intuitively, the "sensing action" `exist_clear(X)_on(Y)` checks if there is a clear block on top of another block. If so, then the top one is unified with `X` and the one beneath it with `Y`.

Apart from the generality of the plans, finding a loopy plan may also be considered as a learning process, and contribute to the solution of large sequential. Typically, it is difficult to find a plan for a large domain. For instance, it may take much longer time to find a plan for a blocks world with 100 blocks than for 3 blocks. However, if we can efficiently *learn* a loopy plan that works for *any* number of blocks from some example problems with only 3 blocks, then the 100-block solution is simply the sequence of actions obtained by executing the loopy plan, which can be done extremely efficiently.

1.3 Logical Foundations

The informal discussion above introduces the main idea of planning and motivates planning with loops in particular. To formally reason about planning domains and represent plans, we introduce, in this subsection, the situation calculus and two robot programming languages based on it.

[3] Abusing Levesque's syntax of robot programs in [15].

1.3.1 The Situation Calculus

First proposed by McCarthy [26] and later refined by Reiter [31], the situation calculus is a logical language for representing and reasoning about dynamical worlds.

The domain of the logic has three sorts, namely, *action* for actions, *situation* for situations, and *object* for everything else. The world evolves with the execution of an action from one situation to another. A situation is represented by a sequence of actions, with S_0 denoting the initial situation where no action has occurred yet. The binary function $do(a, s)$ denotes the situation obtained by executing action a in situation s. In the language, the only function symbols of sort situation are S_0 and $do(a, s)$. Properties that may change their value from situation to situation are called *fluents*. A relational (respectively, functional) fluent is a predicate (respectively, function) that carries a situation term as its last argument. We use $\phi[s]$ to denote the formula obtained from ϕ by restoring the situation argument to all fluents in ϕ with s. Predicates and functions carrying no situational arguments are *rigids*, which have fixed values in all situations.

A basic action theory Σ is needed to characterize any dynamic domain, which consists of the following five sets of axioms [31]:

- Precondition axioms Σ_{pre}, one for each action of the form $Poss(a, s) \equiv \Pi(a, s)$;
- Successor state axioms Σ_{ssa}, one for each fluent of the form (in the case of relational fluent) $F\big(\vec{x}, do(a, s)\big) \equiv \Psi(\vec{x}, a, s)$, to characterize the effects of actions while avoiding the frame problem [25];
- The initial database Σ_0 specifying facts about the initial state;
- Unique names axioms on actions Σ_{una};
- Foundational axioms for situations $\mathcal{FA}$.

Given the action theory $\Sigma = \Sigma_{pre} \cup \Sigma_{ssa} \cup \Sigma_0 \cup \Sigma_{una} \cup \mathcal{FA}$, the task of reasoning about facts in a situation after a sequence of actions has been performed is known as the *projection problem*. One way to solve the projection problem is *regression* [31], which transforms a regressible formula φ that may mention non-initial situations into $\mathcal{R}(\varphi)$ by repeatedly replacing $Poss(a, s)$ with $\Pi(a, s)$ and $F(\vec{x}, do(a, s))$ with $\Psi(vecx, a, s)$. Thus, the only situation term in $\mathcal{R}(\varphi)$ is S_0, and it can be proved that $\Sigma \models \varphi$ if and only if $\Sigma_0 \cup \Sigma_{una} \models \mathcal{R}(\varphi)$ [30]. As a result, with regression, the projection problem can be reduced to first-order theorem proving in the initial theory Σ_0.

1.3.2 Golog

Golog is a logic programming language for high-level control of intelligent agents [14]. It is based on the basic action theory in the situation calculus introduced above,

and offers the possibility to express complex actions with basic control structures, such as **if** ϕ **then** δ_1 **else** δ_2 and **while** ϕ **do** δ, which are similar to the constructs in conventional programming languages. Meanwhile, it allows for the specification of non-deterministic behavior. All of these features are treated as macros which finally expand to formulas in the situation calculus.

The semantics of Golog constructs is formally defined by an abbreviation $Do(\delta, s, s')$ as follows:

1. Primitive actions:

$$Do(a, s, s') \stackrel{def}{=} Poss(a, s) \wedge s' = do(a, s)$$

   where a is a primitive action.

2. Test actions:

$$Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'$$

   where ϕ is a situation-suppressed fluent formula.

3. Sequence:

$$Do(\delta_1; \delta_2, s, s') \stackrel{def}{=} \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s).$$

4. Non-deterministic choice of two actions:

$$Do\big((\delta_1|\delta_2), s, s'\big) \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$

5. Non-deterministic choice of action arguments:

$$Do\big((\pi x)\delta(x), s, s'\big) \stackrel{def}{=} \exists x.Do\big(\delta(x), s, s'\big).$$

6. Non-deterministic iteration: Execute δ zero or more times.

$$\begin{aligned} &Do(\delta^*, s, s') \stackrel{def}{=} \\ &\quad \forall P.\Big\{\forall s_1.P(s_1, s_1) \wedge \forall s_1, s_2, s_3.\big[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)\big]\Big\} \\ &\qquad \supset P(s, s') \end{aligned}$$

Conditionals and while loops can then be defined in terms of the above constructs as

$$\begin{aligned} \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} &\stackrel{def}{=} [\phi?; \delta_1]|[\neg\phi?; \delta_2] \\ \textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} &\stackrel{def}{=} \big[[\phi?; \delta]^*; \neg\phi?\big] \end{aligned}$$

Golog, as introduced here, is a powerful action language, yet it is not directly suitable for representing loopy plans. This is due to its non-deterministic

nature, *i.e.*, Golog programs may need to search different choices, possibly with backtracking. If this non-determinism were allowed, then the Golog program

$$\textbf{while } \neg goal \textbf{ do } (\pi a)[Appropriate(a)?; a] \textbf{ endWhile}$$

would be a generic plan for solving any planning problem. However, running this generic plan is as difficult as the planning task itself [15]. Ideally a "plan" is a procedure for an agent to follow without further deliberation. This motivates the introduction of a plan representation language, where the execution is fully deterministic, so that the plan executer can follow it with no ambiguity or backtracking. The *robot programs* described below is one such language.

1.3.3 Robot Programs

When formalizing planning with sensing, Levesque proposed a simple robot program language whose syntax and execution semantics is inductively defined as follows [15, 16]:

1. **nil** is a robot program executed by doing nothing;
2. for any primitive action A and robot program P, $\textbf{seq}(A, P)$ is a robot program executed by first performing action A, ignoring any sensing result, and then performing the program P;
3. for any sensing action A that has sensing result $R_1, \cdots, R_n$, and any program $P_1, \cdots, P_n$, $\textbf{case}\big(A, [\textbf{if}(R_1, P_1), \cdots, \textbf{if}(R_n, P_n)]\big)$ is a robot program executed by first performing action A, and then performing P_i if the returned sensing result is R_i;
4. if P and Q are robot programs, and P' is the result of replacing some of the occurrences of **nil** by **exit** and the rest by **next**, then $\textbf{loop}(P', Q)$ is a robot program executed by first executing P', and whenever P' ends with **next**, then run P' again, until P' ends with an **exit** at which point Q is executed.

The tree chopping plans in Section 1 can be considered as pretty-printed version of robot programs.

The execution of a robot program generates a *history*, which is a sequence of pairs $(a_i : r_i)$, where the first element, a_i, is the i-th action that the robot program proposes to execute, and the second, r_i, is the corresponding sensing feedback from the environment. In the tree chopping example,

```
[look:up; chop:ok]
```

is a legal history, but a partial one in the sense that the robot program is not run to termination to generate it. In contrast,

```
[look:down; store:ok]
```

and

```
[look:up; chop:ok; look:up; chop:ok; look:down; store:ok]
```

are legal and complete histories.

This robot program language, though simple, is powerful. Indeed, Lin and Levesque has shown that robot programs as defined above are expressive enough to represent any effective controller, provided certain "Turing-machine" actions are provided [20].

1.4 Work Related to Iterative Planning

1.4.1 Program Synthesis

Closely related to iterative planning is program synthesis, which refers to the automatic derivation of a program to meet a given formal specification of its behavior.[24]

Depending on their functionality, the programs are categorized as *applicative* or *imperative*.

An applicative (or functional) program calculates an output based on the given input, producing no side effect during the computation except for the necessary modification of internal data structure. Problems of this type can be expressed as

> Given a constraint on valid input $P(x)$ and the relationship between input and output $R(x, y)$, find a program $f(x)$, such that for any input a that satisfies $P(a)$, the output $z = f(a)$ satisfies $R(a, z)$.

As an example, consider the task of synthesizing a program that reverses a list. Let $List(x)$ be the predicate "x is a list", and $R(x, y)$ be "x is y in reverse order", then the following is a specification for the task to synthesize a program that reverses a list.

> Find a program $rev(x)$ such that for any l satisfying $List(l)$, $z = rev(l)$ satisfies $R(l, z)$.

Imperative programs, on the other hand, may alter the external data structure or to produce other side effects, apart from possibly returning an output of some sort. An example of this type is the Golog program described in the previous section, which produces a sequence of actions as output, and also changes the state of the world.

The loopy plans that we are interested in here fall into the category of imperative programs. Indeed, if the behavior of the loopy plan is fully specified, then iterative planning is simply a special case of imperative program synthesis. Here, a fully specified behavior means a sound and complete axiomatization of the planning domain. That is to say, any model of the theory corresponds to a real world in the planning domain, and any possible world in the planning domain corresponds to a model of the theory.

It may appear that iterative planning, as a special case of program synthesis, is not worth separate research. Unfortunately, this is not true. First, much

of the research on program synthesis is concerned with applicative programs, so imperative program synthesis still needs further investigation. Second, as a planning problem, iterative planning contains rich structures that may be utilized in the algorithm, so that it may become easier to solve than general program synthesis.

1.4.2 Grammar Induction

If the dynamics of the planning domain is fully specified, but no complete axiomatization for the initial possible worlds is given, then the correctness of the resulting program cannot be guaranteed, because the plan can guarantee to work only for the known initial worlds. In this case, iterative planning is a form of inductive inference, and shares some commonality with grammar induction (Section 8.7 of [6]).

The goal of grammar induction is to find the underlying grammar that can generate the observed strings from a language. There can be three types of inputs to the learner: positive examples only, positive and negative examples, or oracle feedback [12]. With positive examples only, the learner is only provided with a sequence of positive examples of the language; for the case of positive and negative examples, the learner is given a sequence of strings, each labeled with whether or not the string belongs to the language; for the oracle feedback type, the learner is allowed to give any string to an oracle, which will then return whether or not the proposed string belongs to the language.

If we consider the set of all legal and complete histories of a planning problem as a formal language where the alphabet consists of actions and sensing outcomes of the problem, and the loopy plan as a grammar that generates or accepts the language, then the iterative planning problem is cast into a grammar induction problem. Furthermore, it falls into the category "oracle feedback", since the action theory serves as an oracle that tells the planner whether a history is legal and achieves the goal.

1.4.3 Repeated-Attempts Problems

Another related type of problems involves probabilistic domains, where the outcome of actions are non-deterministic and outcomes independent, so repeated attempts may be needed for a desired outcome of the action to occur. Unlike in the previous two subsections where the seemingly unrelated problems have a deep connection with iterative planning, problems of the repeated-attempt style are less close to our notion of iterative planning than they may first appear.

One special case of the repeated-attempts category involves probability, which is studied in-depth in decision-theoretic planning. For example, when picking up a block on the table, there is possibility p that the block is successfully picked up and $1 - p$ that it remains on the table [17]. So, if one wants to guarantee the block in hand, the pick-up action will need to be performed repeatedly. A similar problem is in the omelette domain where the chance of getting a good egg is a fixed number [3]. In these examples, the repetition of

actions have independent outcomes, but such independence does not exist in deterministic domains like tree chopping or clearing blocks.

Cimatti *et al.* proposed three criteria for plan validity in non-deterministic domains [5]. A *weak* plan is one that may achieve the goal, but not guaranteed to do so; a *strong* plan is one that always achieves the goal, no matter what output each action generates; a *strong cyclic* plan is one that guarantees to achieve the goal under a "fairness assumption" and possibly after repeated trial and error. According to this definition, all repeated-attempts problems that need loopy plans can at best have strong cyclic, and cannot have strong solutions. This is due to the outcome-independence assumption that they insist on the action definition. The iterative planning problems that we are interested in, in contrast, may have "strong" solutions even though loops are needed in them.

2 Existing Approaches to Iterative Planning

In this section, I will survey existing work on iterative planning, which falls roughly into two categories, namely, *deductive* and *non-deductive* (inductive) approaches.

2.1 Deductive Approaches

Deductive approaches to planning typically generate a plan as a by-product of proving a mathematical theorem. A short survey on deductive planning is conducted by Biundo in [2]. However, most work in this category does not attempt to generate loopy or recursive plans, but here are a few exceptions.

In an early work, Manna and Waldinger proposed a tableau-based sequent calculus to deductively synthesize applicative programs [22]. Domain axioms and the program specification are represented by sequents, along with a mathematical induction rule. Each sequent is a three-column table representing assertions, goals and outputs. For example, the program specification "find a program $f(x)$ such that for all input a satisfying constraint $P(a)$, the output $f(a)$ satisfies $R(a, f(a))$" can be represented by

assertions	goals	outputs $f(a)$
$P(a)$	$R(a, z)$	z

The meaning of a sequent is that if all instances of the assertions are true, then at least one of the goals is true. Associated with this sequent notation is a set of derivation rules, which preserves correctness and can be used to obtain new sequents. The deduction terminates whenever the assertion *false* or the goal *true* is derived. At this point, the content in the "outputs" column of this sequent is a program that satisfies the problem specification.

Manna and Waldinger later extended this tableau-based approach with a variant of the situation calculus for representing actions dynamics, so as to synthesize recursive plans in planning domains [23]. Recursion is introduced

by the well-founded induction rule, *i.e.*, an induction rule over a well-founded relation. This works well when there is a perfect match between the current goal, plan and the induction hypothesis. However, when the matching is close but not perfect, generalization is needed to prove a stronger induction hypothesis. Unfortunately, how this generalization can be automated remains unclear.

With a similar goal, Stephan and Biundo proposed a deduction-based refinement planning approach based on a temporal planning logic [38]. Their idea of refinement planning is to start from a non-constructive problem specification, and gradually refine it to generate an executable plan. Since representations of specifications and plans are on the same linguistic level in temporal planning logic, the object of the refinement is a mixed representation of the two, where some part of the specification is replaced by executable plan segments in each of the refinement steps. Due to the soundness of the refinement rule, the plan that they finally derive is guaranteed to be provably correct.

Unlike Manna and Waldinger's approach [23], loops are not introduced in the refinement steps, but instead prefabricated in the initial abstract problem specification. As a result, this approach falls into the category of refining human-designed abstract loopy plans, instead of creating them automatically.

In both approaches above, iterative planning is an interactive process, where human is needed to provide either the strengthened loop invariant or the high level plan. Indeed, no complete, fully automatic procedure exists for deductive iterative planning, since plans with recursion are Turing complete. However, if we are only after an incomplete algorithm which can solve interesting special cases, such deductive approach may still work, armed with clever heuristics.

For example, Magnusson and Doherty recently proposed a deductive planning framework for maintenance-goal problems in temporal action logic [21]. In order to get around the invariant strengthening problem also faced by Manna and Waldinger, they incorporated a regularity heuristic and a synchronization heuristic to help the deductive reasoner find useful invariants. 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. Magnusson and Doherty proved that both the heuristics and the induction rule are sound, *i.e.*, whenever their algorithm outputs a plan, it is guaranteed to be correct. They also showed that their planner solves a practical surveillance problem based on this approach.

2.2 Non-deductive Approaches

In this section, I will review work on non-deductive approaches to iterative planning. These methods are typically based on identifying repeating patterns in loopless plans, and then greedily form loops from them, so they sidestep the reasoning about induction hypothesis and loop termination. As a result, non-deductive approaches are typically more efficient than their deductive counterparts, but meanwhile offer a much weaker correctness guarantee.

2.2.1 Generate and Test

In a recent work [16], Levesque implemented a planner called KPLANNER that automatically identifies loopy plans represented with robot programs defined in Section 1.3.3.

KPLANNER is designed for automatically finding robots program for planning problems with a "planning parameter". This parameter is a numerical fluent in the planning problem that is not known or even bounded at planning time, and thus requires the existence of loops in the plan.

To solve this type of problems, KPLANNER repeatedly shifts between a generation and a testing phase. In the generation stage, the planner is provided with a small constant N_1 called the *generation bound* as the value of the planning parameter, and is asked to exhaustively search for loopy plans that works for this value. This is done by first finding a conditional plan that works for N_1. Then, the planner attempts to wind the conditional plan into a loopy one, and whenever the winding succeeds, the resulting plan is passed to the testing phase.

During the testing phase, the plan is tested against a larger constant N_2 called the *testing bound*. If the plan fails in this phase, then KPLANNER enters the generation phase again, and continues with the search for loopy plans that works for N_1, until some plan passes the testing phase, at which point the plan is returned as the output.

As we can see, the output plan is not guaranteed to work for all possible values of the planning parameter, but instead only for N_1 and N_2. However, for many practical problems as shown in the paper, the resulting plan is indeed a valid solution that works for all values. Levesque also proved as a theorem that if the planning problem satisfies certain constraints, then the plan generated by KPLANNER is guaranteed to be correct for all possible values of the planning parameter.

2.2.2 Identifying Regularity in Partial-Order Plans

In contrast to using a planning parameter, Winner and Veloso developed algorithms for finding program-like "domain specific planners" (dsPlanners), for a class of ADL problems [41, 42].

The input to their first algorithm, DISTILL [41], is the domain specification, along with solutions to some example problems, which are processed sequentially to revise the current dsPlanner starting from an empty program. The output of DISTILL is a compact conditional plan that solves all the examples and likely other similar problems.

To accommodate an example solution, the plan is first parametrized to best match the current dsPlanner, and then converted into a new dsplanner by introducing `if` statements for selecting objects for the parameters. The conditions for the if statement are the initial, current and goal state terms that are relevant to the plan, which can be obtained using the minimal annotated consistent partial ordering [40] of the observed plan. This partial ordering information is also stored for later merging the existing dsPlanner with the newly obtained one.

To merge the two dsPlanners, the DISTILL algorithm searches through each of the `if` statement in the current program, and tries to find a matching in the new dsPlanner. If such a matching is found, then the two `if` statements are combined; otherwise, a new `if` statement is appended to the end of the current dsPlanner.

DISTILL, in its simplest form, cannot introduce loops to the resulting dsPlanners. To overcome this drawback, Winner and Veloso later proposed a variant of the algorithm called LoopDISTILL, which is able to automatically identify parallel and serial loops [42].

The basic idea is similar to that of DISTILL. Here, instead of finding a dsPlanner for the whole parametrized plan, it identifies the largest matching subplan, and convert the repeating occurrences of these subplans into a loop. This procedure is repeated greedily, until no further loops can be formed. Unlike in the DISTILL algorithm, no procedure for merging dsPlanners with loops is given in the paper, so LoopDISTILL essentially only uses one single plan for generalization, and it is unclear how multiple example plans can be taken into account.

Winner and Veloso claim that LoopDISTILL can synthesize dsPlanners with complex but non-nested loops.

2.2.3 Using Role-Based Object Abstraction

With a similar goal to find loopy plans for a class of STRIPS-like problems, Srivastava *et al.* proposes to use state aggregation to group objects in the same role into equivalence classes, and obtain an abstract state representation [36].

In their formalism, a *role* is defined as a conjunction of literals consisting of every abstraction (unary) predicate or its negation. As a result, a domain with N unary predicates can have at most 2^N roles and thus at most 2^N abstract objects, no matter how many actual objects exist in the domain.

To obtain a loopy plan, their algorithm only needs one concrete plan containing sufficient un-rollings of some loops. From this plan, all the actual objects are replaced by their corresponding abstract objects. At this point, the repeating pattern of the plan becomes obvious, and loops can be obtained by folding the repeating sections in the abstract plan [37].

Like Winner and Veloso's approach, their algorithm can only detect non-nested loops. However, Srivastava *et al.* require the stronger assumption that independent actions within each repetition must happen in the same order so that loops can be detected, which is less general than Winner and Veloso's partial order representation. Like LoopDISTILL, it is unclear how multiple example plans can help improve the generality or accuracy of the resulting loopy plan in Srivastava *et al.*

On the other hand, the role-based formalism has the advantage that it can identify sufficient conditions for the resulting loopy plan to be provably correct. This is done by back-propagating constraints on three-valued structures in the abstract space [36]. As a theorem, they proved that extended-LL (linked-list) domains are amenable to back propagation, and thus loopy plans in these

domains found by their algorithm are always correct.

2.2.4 Explanation-Based Generalization

Shavlik's BAGGER2 views the problem of generating recursive and iterative concepts as an explanation-based learning problem extended with the ability of "generalizing to N" [34], and the system applies naturally in planning domains such as blocks world. Based on a similar idea, Schmid and Wysotzki learn iterative macro operators for a planning domain [33] by inductive program synthesis [13]. Unlike BAGGER2, where the generalization is done over the proof tree of the background theory, they assume basic data type structures (natural numbers, lists, sets, *etc.*), explore problems of small complexity, and generate finite programs from the universal plans of these problems, whose syntactical structure is then explored to generate loops.

3 Theories on Correctness

3.1 Finite Model Checking

Lin recently proposed the concept of finitely-verifiable theories [18]. Intuitively, a finitely-verifiable class of sentenses has the property that whether or not a sentence in the class is a theorem can be checked with respect to a finite set of models of the theory.

This idea, applied to the action domain, can be used to prove goal achievability in a set of initial states [19]. In particular, to see if a loopy plan solves all the problems in a finitely-verifiable planning domain, one can use Lin's result, and identify the finite set of models that is sufficient for the judgment. Then, the correctness of the loopy plan can be concluded by model checking. As a result, Lin's result serves as an alternative way of proving plan correctness, and may be an especially useful supplement to the non-deductive planning approaches where correctness is not guaranteed by the planner itself.

3.2 Identification in the Limit

When using Lin's notion of finite-verifiability in planning domains, meta-theoretic analysis must be performed over the models characterizing legal initial states [19].

When this characterization is not available, it is not possible to obtain similar correctness result. This happens, for example, under the common paradigm where a list of possible initial states are given, and the planner does not have clue about validity of other initial states.

As Caldon and Martin pointed out [4], this setting can be understood with Gold's learnability model of identification in the limit [12]. It is a grammar learning model where the learner is required to learn a grammar from an infinite supply of sentences generated by the grammar. After seeing each new sentence in the sequence, the learner knows immediately whether its current hypothesis (of the grammar) is wrong. If it is, then the learner may revise its hypothesis,

before observing the next example. A grammar is then considered learnable in this model, if the learner can make at most finite mind-changes before its hypothesis is correct.

Although the connection has be identified, how this learnability model can contribute to theoretical results in planning with loops, such as proving correctness properties, remains an interesting question and left to be explored.

4 Future Work

As can be seen from our discussion in Section 2, iterative planning remains a challenging task in that deductive approaches are slow and requires human interaction, whereas non-deductive approaches typically have limited applicability. Finding an efficient and more general algorithm is a candidate direction for future work.

It is also interesting to apply the idea of iterative planning to other problem domains. For example, in 2008, the International Planning Competition (IPC) introduced a new learning track [39], where each participating planner is first given a few domains and a dozen example problems in each of them in the learning phase. The planner then can take one week's time to extract knowledge from these examples, which can be used in the testing phase to help improve efficiency and/or plan quality on new problems in the same domain.

This paradigm is very close to the non-deductive iterative planning themes that we covered in Section 2.2. It is likely that results in iterative planning can be applied or otherwise contribute to more efficiently solving the competition domains or improve plan quality.

On the theoretical side, correctness guarantee is of of the areas to be explored. As mentioned, many of the non-deductive approaches, including KPLANNER, do not have a strong guarantee on the correctness of the generated plan. It is thus important to identify classes of problems that these algorithms guarantee to work. Besides, identifying the complexity of the planning task is also a promising yet challenging line of research.

5 Conclusion

This paper serves as a short survey on the existing approaches to planning with loops or recursions. Although this is not a new topic, progress has been limited with the deductive approach in the past three decades, due to its intrinsic difficulty. Recently, however, there is a revitalization of this topic, and many non-deductive alternatives are proposed and analyzed. Further investigation and extension of these ideas remains an interesting research direction, and new theories about iterative planning may be yet to be explored.

References

[1] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI*, pages 473–478, 2001.

[2] Susanne Biundo. Present-day deductive planning. *Current Trends in AI Planning*, pages 1–5, 1994.

[3] Blai Bonet and Héctor Geffner. GPT: A tool for planning with uncertainty and partial information. In *IJCAI-01 Workshop on Planning with Uncertainty and Incomplete Information*, pages 82–87, Seattle, WA, 2001.

[4] P. Caldon and E. Martin. Learning a plan in the limit. In *Proceedings of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2007.

[5] 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.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, 2001.

[7] R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[8] Website for International Planning Competition. http://ipc.icaps-conference.org, 2008.

[9] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 2003.

[10] A. Gerevini and D. Long. Plan constraints and preferences in PDDL 3. Technical report, University of Brescia, Italy, 2005.

[11] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – the planning domain definition language. Technical report, Yale University, 1998.

[12] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[13] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: an explanation based generalization approach. *Journal of Machine Learning Research*, 7:429 – 454, 2006.

[14] H. Levesque, Reiter R., Y. Lesperance, Lin F., and Scherl R. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

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

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

[17] Peter Haddawy Liem Ngo. Representing iterative loops for decision-theoretic planning. In *Working Notes of the AAAI Spring Symposium on Extending Theories of Action*, 1995.

[18] F. Lin. Finitely-varifiable classes of sentences. In *Commonsense*, 2007.

[19] F. Lin. Proving goal achievability. In *KR*, 2008.

[20] F. Lin and H.J. Levesque. What robots can do: robot programs and effective achievability. *Artificial Intelligence*, 101(1-2):201–226, 1998.

[21] Martin Magnusson and Patrick Doherty. Deductive planning with inductive loops. In *AAAI-08*, 2008.

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

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

[24] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18:674–704, 1992.

[25] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[26] John McCarthy. Situations, actions, and causal laws. Technical report, Stanford Articial Intelligence Project, Stanford University, 1963.

[27] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Principles of Knowledge Representation and Reasoning*, 1989.

[28] R. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling*, pages 212–221, Menlo Park, CA, 2002. AAAI Press.

[29] 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*, pages 2–11, Menlo Park, CA, June 2004. AAAI Press.

[30] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999.

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

[32] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[33] U. Schmid and F. Wysotzki. Applying inductive program synthesis to macro learning. In *Artificial Intelligence Planning Systems*, pages 371–378, 2000.

[34] J.W. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5(1):39–70, 1990.

[35] D. Smith and D. Weld. Conformant graphplan. In *AAAI*, 1998.

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

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

[38] W. Stephan and S. Biundo. Deduction-based refinment planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, 1996.

[39] Learning track of the International Planning Competition. http://eecs.oregonstate.edu/ipc-learn, 2008.

[40] E. Winner and M. Veloso. Automatically acquiring planning templates from example plans. In *Proceedings of Artificial Intelligence Planning Systems*, 2002.

[41] E. Winner and M. Veloso. DISTILL: Towards learning domain-specific planners by example. In *Proceedings of International Conference on Machine Learning*, 2003.

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