Conditionalization: Adapting Forward-Chaining Planners to Partially Observable Environments

Ugur Kuter **Dana Nau** **Elnatan Reisner**
Department of Computer Science
and Institute for Systems Research
University of Maryland,
College Park, MD 20742, USA
{ukuter,nau,elnatan}@cs.umd.edu

Robert Goldman
Smart Information Flow Technologies
(d/b/a SIFT, LLC)
211 N. First St., Suite 300
Minneapolis, MN 55401, USA
rpgoldman@SIFT.info

Abstract

We provide a general way to take forward-chaining planners for classical planning domains and *conditionalize* them, i.e., adapt them to generate policies for partially observable planning domains.

For domain-configurable planners such as SHOP2, TLPlan, and TALplanner, our generalization technique preserves the ability to use domain knowledge to achieve highly efficient planning. We demonstrate this experimentally in two problem domains. The experiments compare PKS and MBP (two existing planners for partially observable planning) and CondSHOP2, a version of the HTN planner SHOP2 created by applying our conditionalization method.

To our surprise, PKS and MBP could solve only the simplest problems in our test domains. In contrast, CondSHOP2 solved all of the test problems quite easily. This suggests that the ability to use domain knowledge may be not just desirable but indeed essential for solving large problems in partially observable domains.

Introduction

One of the biggest limitations of classical AI planning is the assumption that the entire initial state is known at planning time. A more realistic assumption is that the planner only knows part of this information at planning time, and the rest must be acquired at plan-execution time, through observations or queries. This means the planner must construct a plan that includes both sensing actions and conditional execution of the actions in the plan. Several planning algorithms have been formulated to do this; examples include PKS (Petrick & Bacchus 2002; 2004) and MBP (Bertoli *et al.* 2001b).

This paper provides the following contributions:

- We provide a general way to take forward planners for classical planning domains and *conditionalize* them, i.e., adapt them to generate policies for partially observable planning domains. For domain-configurable planners such as SHOP2 (Nau *et al.* 2003), TLPlan (Bacchus & Kabanza 2000), and TALplanner (Kvarnström & Doherty 2001), our generalization technique preserves their ability to use domain knowledge to achieve highly efficient planning.
- We provide experimental comparisons of PKS and MBP (two previous planners for partially observable planning domains) with CondSHOP2, a version of the HTN planner SHOP2 created by applying our conditionalization method. The experimental domains include adaptations of the Unix domain that the authors of PKS used in their experiments, and the Robot Navigation domain that the authors of MBP used in their experiments.
- In our experiments, CondSHOP2 outperformed PKS and MBP; but it did so much more dramatically than we had anticipated: PKS and MBP could solve only the simplest problems in our test domains, but CondSHOP2 solved all of the problems quite easily. This suggests that the ability to use domain knowledge may be not only desirable but indeed essential in order to solve large problems in partially observable planning domains.

Formalism

Chapter 2 of (Ghallab, Nau, & Traverso 2004) gives two formalisms for classical planning: the *classical* and *state-variable* representations. The two representations are completely equivalent, in that each can be transformed to the other in linear time. Our formalism is based on the state-variable representation, which can be summarized as follows. There is a finite set of state-variable symbols V and a finite set of constant symbols C. A state-variable assignment is a pair (v, c), where $v \in V$ and $c \in C$. A *state* is a set s of state-variable assignments that includes exactly one assignment for each $v \in V$. A *planning operator* o includes a set of preconditions $pre(o)$ and a set of effects $eff(o)$, both of which are sets of state-variable assignments. An *action* is any ground instance of a planning operator. A *goal* is a set g of state-variable assignments that includes at most one assignment for each $v \in V$.

We can extend the state-variable representation to include partial observability. Due to space constraints we must omit the details, but here's a quick summary:

- A *partial state* is a set s of pairs (v, x) that includes exactly one pair for each $v \in V$, where each pair (v, x) is either a state-variable assignment or an expression

of the form (v, unk) to indicate that v's value isn't known. A *completion* of s is any complete state that contains all of the state-variable assignments in s. $K(s)$ is the set of all completions of s.

- For each $v \in V$, there may or may not be a *sensing action observe*(v) that assigns to v whatever value it senses at execution time.
- The ordinary non-sensing actions are the same as before. If an action a is applicable to a partial state s, then it is applicable to every completion of s.
- Instead of a plan, we have a policy π over partial states. To avoid exponential blowup, we represent π as a *state-based* policy, i.e., a set of pairs $\pi = \{(S_i, a_i)\}_{i=1}^{k}$, where S_i is a set of partial states and a_i is the action to execute in each of those partial states.
- A policy π solves a partially observable planning problem P whose initial state is s_0 if π solves every planning problem that can be produced from P by replacing s_0 with a completion of s_0.
- A *partially observable classical (POC)* planning problem is a 4-tuple $P = (L, O, N, s_0, g)$, where L is the planning language, O is the set of operators, N is the set of sensing actions, s_0 is the *initial* partial state, and g is a set of state-variable assignments. A *completion* of P is any classical planning problem $P' = (L, O, N, s'_0, g)$ such that s'_0 is a completion of s. $K(P)$ is the set of all completions of P.
- We use the obvious generalization of the state-transition function γ to partial states, and usual definition of a policy's *execution structure*. The *execution structure* for π is a directed graph Σ_π whose nodes are all of the partial states that can be reached by executing actions in π, and whose edges represent the state transitions for those actions. If there is a path in Σ_π from s_1 to s_2, then we say that s_1 is a π-ancestor of s_2 and s_2 is a π-descendant of s_1. If one of the π-descendants of a state s is itself, then we say that s is a cyclic state in π.

Theorem 1 *π is a solution for P iff π is a solution for every completion of P.* (Proof omitted due to space constraints.)

Conditionalizing Forward Planners

We now describe a general way to take forward-chaining planners for classical planning, and *conditionalize* them, i.e., translate them into planners that find solutions to POC planning problems. The basic idea is to take an abstract planning procedure that includes the classical planners as instances, and translate it into a procedure that can solve POC planning problems. The same translation will then apply to all of the instances.

As our abstraction of forward-chaining classical planners, we use the FCP procedure that appeared in (Kuter & Nau 2004). This procedure is shown in Figure 1. FCP is general enough to include many (perhaps most) forward-chaining planners as instances—including, for example, several well-known domain-configurable planners, such as TLPlan (Bacchus & Kabanza 2000), SHOP2 (Nau *et al.* 2003), and TALplanner (Kvarnström & Doherty 2001).

```
Procedure FCP(L, O, s_0, g, α);
  π ← ∅; s ← s_0
  loop
    if s satisfies g then return(π)
    A ← {(s, a) | a is a ground instance of an operator
          in O, a is applicable to s, and a ∈ α(s)}
    if A = ∅ then return(failure)
    nondeterministically choose (s, a) ∈ A
    π ← π ∪ {(s, a)}
    s ← γ(s, a)
```

Figure 1: FCP, the abstract forward-chaining classical planner in (Kuter & Nau 2004).

```
Procedure CondFCP(L, O, N, s_0, g, α)
  π ← ∅; S ← {s_0};
  loop
    if S = ∅ then return(π)
    select a partial state s ∈ S and remove it from S
    if s does not satisfy g then
      if s ∉ S_π then
        A ← {(s, a) | a is a ground instance of an
              operator in O ∪ N, a is applicable
              to s, and a ∈ α(s)}
        if A = ∅ then return(failure)
        nondeterministically choose (s, a) ∈ A
        if a is a sensing action then
          let v be the state-variable for a
          S' ← {(s − {(v, z)|z ≠ c}) ∪ {(v, c)} |
                    c ∈ range(v)}
          S ← S ∪ S'
        else
          S ← S ∪ {γ(s, a)}
        π ← π ∪ {(s, a)}
      else if s is a cyclic state in π then
        return(failure)
```

Figure 2: Our conditionalization of FCP. The underlines indicate how the FCP coding is embedded in CondFCP.

As shown in Figure 1, FCP starts with an initial state s_0, and explores other states by successively choosing and applying planning operators until it reaches a state that satisfies the goal formula g. The *pruning function* $\alpha(s)$ is used to prune the search space. Rather than trying all of the actions applicable to s, FCP only tries the ones that are both applicable and returned by α.

For example, in SHOP2, the search is controlled by a set of *methods* that decompose *tasks* into subtasks, and $\alpha(s)$ is the set of all actions a applicable to s such that a can be produced by applying methods to the current task network. In TLPlan, the search is controlled by a formula ϕ written in *Linear Temporal Logic (LTL)*. For each state s, $\alpha(s)$ is the set of all actions a applica-

ble to s such that the successor state $\gamma(s, a)$ satisfies a *progressed* formula, $Progress(s, \phi)$.

Figure 2 shows CondFCP, our conditionalized version of FCP. If a planning algorithm $\mathcal{A}$ can be described as an instance of FCP, then the conditionalization of $\mathcal{A}$ is Cond$\mathcal{A}$, the corresponding instance of CondFCP. Using this technique, we can easily write conditionalizations of TLPlan, SHOP2, TALplanner, and several other forward planners that are instances of FCP.

Like FCP, CondFCP plans forward from S_0, but the success criterion is more complicated. In FCP, a plan π is a solution if its final state satisfies the goal condition g. In CondFCP, in order for a policy to be a solution, every path in π's execution structure must lead to a partial state that satisfies g. Thus, CondFCP essentially performs an AND-OR search (Nilsson 1980) over a space of partial states, where the AND branches are represented by the set S and the OR-branches correspond to the choice of the actions.

Every time CondFCP considers a partial state s from S for expansion, it first checks whether s satisfies the goal g. If so, it ignores s and selects another partial state from S. If s does not satisfy the goal, then CondFCP chooses an action for s using its pruning function α. The pruning function α generates a set of *acceptable* actions that are applicable in s. More specifically, in a partial state s, α either returns a ground instance of a planning operator (i.e., an action in the classical planning sense), or it returns a sensing action to be applied in s. In the former case, CondFCP simply inserts the successor state $\gamma(s, a)$ generated by applying a in s. The latter, on the other hand, means that the current policy being formulated by CondFCP needs to gather information about a state variable at this point during its future execution. Thus, CondFCP inserts into S all possible partial states that are successors of s that might be generated by applying the sensing action in s during execution. This makes the planner "execution robust" in the sense that the planner always generates a plan that specifies what to do for each possible outcome of a sensing action during execution.

CondFCP performs successive iterations until there are no partial states left in S. This means that from every partial state that is reachable from the initial state s_0 by applying the actions in π, a goal state is reachable; thus, π is a solution to the input POC planning problem and CondFCP returns π.

CondFCP's pruning function is analogous to FCP's, and the pruning function for a classical planning domain can generally be used in partially observable versions of the domain. As an example, recall that for TLPlan, the pruning function α for a planning problem $\mathcal{P}$ returns all actions applicable to s such that $\gamma(s, a)$ satisfies $Progress(s, \phi)$, where ϕ is a search-control formula. If $\mathcal{P}'$ is a partially observable version of $\mathcal{P}$, then α could work for $\mathcal{P}'$ as follows. If the formula ϕ can be evaluated in a partial state s, then α returns all actions applicable to s as above. If the planner needs additional information about a state variable v to evaluate ϕ in s, then α specifies v, if there is a sensing action that corresponds to it. Then, for each partial state s' that is a successor of s given that sensing action, α returns all actions applicable in s' such that $\gamma(s', a)$ satisfies $Progress(s', \phi)$.

It's not hard to show that FCP and CondFCP are both sound (i.e., they don't return plans that are not solutions), and conditionally complete (they can find every solution whose actions are returned the pruning function). The following theorem provides a bound on CondFCP's time complexity; we omit the proof due to space constraints.

Theorem 2 *Let $\mathcal{A}$ be an instance of FCP, and $P = (L, O, N, s_0, g)$ be a POC planning problem. Suppose we run* Cond$\mathcal{A}$ *on P with a pruning function α that runs in polynomial time, and* Cond$\mathcal{A}$ *returns a solution π. Then* Cond$\mathcal{A}$*'s running time is polynomial in $|\Sigma_\pi|$, where $|\Sigma_\pi|$ is the size of the execution structure of the solution for π.* (Proof omitted due to space.)

It can be shown that for some deterministic implementations of CondFCP, the worst-case running time is polynomially bounded by FCP's worst-case running time, when both running times are expressed as functions of the size of the returned policy. CondSHOP2, our conditionalized version of SHOP2, is such a case.

Experimental Evaluation

When writing pruning functions—e.g., the HTN methods for a planner like SHOP2 or the LTL control rules for a planner like TLPlan—the usual strategy is to try to minimize the amount of backtracking that will need to be done, by pruning all but a few of the applicable actions.

In classical planning problems, this approach has been quite successful. For example, (Bacchus & Kabanza 2000) and (Nau *et al.* 2003) describe problem domains in which the planner goes directly to a solution with almost no backtracking—effectively turning what might otherwise have been an exponential-time search into a polynomial-time search instead.

This section experimentally examines how well the same technique works on partially observable planning problems. For our experiments, we implemented CondSHOP2, the conditionalizaton of the HTN planner SHOP2 (Nau *et al.* 2003). We compared its performance and scalability with PKS (Petrick & Bacchus 2002; 2004) and MBP (Bertoli *et al.* 2001b), two planners that we believe to represent the state of the art in planners for partially observable planning problems. We ran all of our experiments on an 2.16 GHz Intel Core Duo MacBook laptop computer, running Linux Fedora Core 6 via a virtual machine with 256MB memory.

Unix domain. We compared PKS and CondSHOP2 on a modified version of the Unix domain on which PKS was tested in (Petrick & Bacchus 2002). In Petrick & Bacchus's planning problems, the only Unix operations available to the planner were `cd`, `cdup`, and `ls`; and the

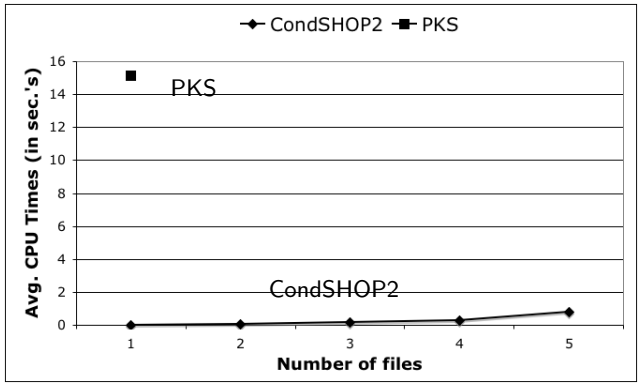

Figure 3: Average running times of CondSHOP2 and PKS in the Unix domain, as a function of the number of files. Only one data point is shown for PKS because it exceeded our 30-minute time limit on larger problems.

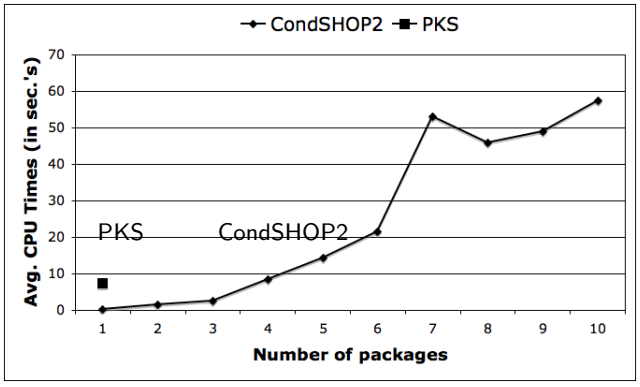

Figure 4: Average running times of the three planners in partially observable Robot Navigation problems, as a function of the number of packages. PKS has only one data point because it exceeded our 30-minute time limit on larger problems. MBP is not shown because it ran out of memory on all problem sizes.

task was to look through a directory structure to find the locations of various files. In our planning problems, the task was to locate the files and move them to various destinations, and the available Unix operations were `cd`, `cdup`, `ls`, `mv`, `cp`, and `rm`.

Since CondSHOP2 is an HTN planner, its pruning function α is computed via a set of HTN methods. We wrote HTN methods to induce an ordering $f_1, \ldots, f_n$ on the files to be moved. When none of the files has been moved, α returns the actions to find f_1 and move it; when f_1 has been moved, α returns the actions to find f_2 and move it; and so forth. Dealing with the files in this order avoids the exponential blowup that would occur if the planner examined all of the orderings in which the actions could be done.

Figure 3 shows the results of our first experiment. Each data point is the average of 20 random problems. If a planner did not return a solution for one or more problems within the alloted time (30 minutes per problem) or overflowed the available memory, we omitted the corresponding data point. For PKS, we only have a data point for $n = 1$, because for $n > 1$, PKS was unable to solve problems within the alloted time.

Partially observable Robot Navigation domain. We compared PKS, MBP, and CondSHOP2 on a modified version of the Robot Navigation domain. The original Robot Navigation domain provided a fully observable nondeterministic environment for tests of MBP (Bertoli *et al.* 2001b), ND-SHOP2 (Kuter & Nau 2004), and other planners. It consists of a building with 8 rooms connected by 7 doors. In the building, there is a robot and there are a number of packages in various rooms. The robot is responsible for delivering packages from their initial locations to their final locations by opening and closing doors, moving between rooms, and picking up and putting down the packages. The robot can hold at most one package at any time.

To make the domain deterministic, we omitted the "kid doors" (doors that could open and close at random). To make it partially observable, we assumed that initially, the robot doesn't know what packages are in each room and where those packages need to go.

Figure 4 shows the results of our comparison of all three planners. As before, each data point is an average of 20 random problems. As before, if a planner did not return a solution for one or more problems within the alloted time (30 minutes per problem) or overflowed the available memory, we omitted the corresponding data point. PKS's only data point is for one package, because it ran out of time in problems where there was more than one package. There are no data points for MBP because it ran out of memory regardless of the number of packages.

In contrast, CondSHOP could solve all of the problems very quickly. The reason for its fast performance is again due to the pruning function α. As before, α was computed via HTN decomposition, and we wrote HTN methods that enforced a fixed order in which CondSHOP2 would seek and deliver the packages. As before, this avoided an exponential blowup in the size of the search space.

A very simple domain. Although we had expected PKS and MBP to do worse than CondSHOP2 in the above two experiments, we were quite surprised that they failed so completely. To investigate this further, we created a very simple version of the partially observable robot navigation domain, in which (1) there is only one package to deliver, (2) the package's destination is known in the initial state, and (3) rather than allowing the package's initial location to be any of the 8 rooms in the domain, we restricted the initial location to be any of just r different rooms, where $2 \leq r \leq 8$.

Figure 5 shows the performance of all three planners as a function of r in this very simple domain. Each data

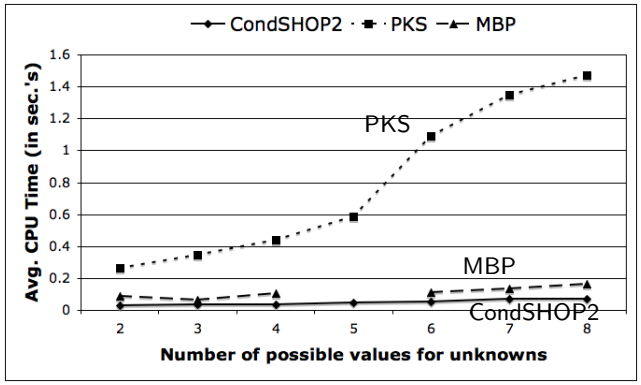

Figure 5: Average running times of the three planners in a much simpler version of the Robot-Navigation problem with only one package, as a function of the number r of possible rooms the package might be in. MBP has no data point for $r = 5$ because it ran out of memory on one of those problems.

point is an average of 20 random planning problems. PKS solved all the problems, but with much larger running times than MBP and CondSHOP2. MBP's running times were closer to CondSHOP2's, but the data point for $r = 5$ is missing for MBP because it exceeded the available memory for one of these problems.

Discussion on the Experimental Results.

Our experimental results suggest that the ability for a planner to use domain knowledge may be not only desirable, but indeed essential, to solve planning problems in partially observable domains. Traditionally, domain knowledge has been used in planners such as SHOP2, TLplan, and TALplanner in order to prune their search space; i.e., to prune actions that are irrelevant to the solution(s) of a planning problem. Under partial-observability, we observed that this property still holds and the conditional planner, i.e., CondSHOP2 in this case, would benefit from it since it has the ability to exploit such knowledge due to our generalization technique. However, we also observed that this ability not only enables the planner to prune actions, but also allows it to choose what observation (i.e., sensing) needs to be done in a solution plan generated by the planner. In other words, the ability to use domain knowledge enabled CondSHOP2 in these experiments to prune the irrelevant observations/sensing from the plans. We hypothesize that this made the planner very efficient in comparison to PKS and MBP because both PKS and MBP, the best of our understanding, explore search space where the branching factor is not only determined by the actions applicable in a state but also the number of possible observations/sensing that can be done in that state.

Some additional remarks on the performance of PKS and MBP in these experiments also are in order. Note that, although PKS and MBP have been tested on *a* Unix domain and *a* Robot Navigation domain in previous studies on these planners (Petrick & Bacchus 2002; 2004; Bertoli *et al.* 2001a; Pistore, Bettin, & Traverso 2001), these domains are much simpler than the ones we considered in this paper. In particular, to the best of our knowledge, PKS was previously tested on simple variants of Unix domain that include only the relevant actions for the planning problems: e.g., in (Petrick & Bacchus 2002), the Unix domain had only three actions (namely `cd`, `cdup`, and `ls`); and the task was to look through a directory structure to find the locations of various files as we mentioned above. Our Unix domain above, on the other hand, involves moving and copying files around the directory structure as well, which increases the size of the search space for PKS exponentially. Similarly, although previous works report performance results for MBP in the Robot Navigation domain, those result were on the planning algorithms implemented in MBP for *fully-observable planning in nondeterministic domains*. To the best of our knowledge, the partial-observable planning algorithms in MBP have not been tested in the Robot Navigation domain above — note that the studies partial-observable planning with MBP reported in (Bertoli *et al.* 2001b; 2006) mentioned a robot domain, however that is a different and simple domain for illustration purposes only.

However, we also noticed that the version of PKS that was available on the Web at the time we were doing these experiments was not able to handle exclusive-OR knowledge properly. We believe that this was a major source of inefficiency for PKS in our experiments since we depend on such knowledge in our domain encodings. A personal communication with the authors of PKS revealed that this issue will be resolved in the next version of PKS. We are planning to conduct experiments with that new version when it is available.

Related Work

CNLP (Peot & Smith 1992) is a partial-order causal-link (POCL) planner (a variant of SNLP) that generates conditional plans. The actions in its plans are annotated with contexts and reasons. Contexts indicate what states of the world are possible at particular points in the plan. Reasons are the goals which this action helps achieve. Bookkeeping for contexts and reasons is more complex than in our approach, because the steps are only partially-ordered. Like most of the early conditional planners, CNLP does not make a clear distinction between the state of the world and the agents' knowledge of the world state, representing only the latter explicitly. Observing a state variable, v is modeled as a non-deterministic action whose precondition is *unk(v)* (there may be others, as well), and which branches into one state where v holds, and one where *not(p)* holds. For limitations of this approach to POC planning, see (Goldman & Boddy 1994b). In the published version of CNLP, ordering constraints from one context could "bleed over" into other contexts, making the the algorithm as published incomplete: it misses

some plans because unnecessary ordering relations can cause it to fail, where the orderings were on different contexts. We understand that Peot's thesis includes a revised version of CNLP that fixes this problem, but the revised version has not been published elsewhere.

Cassandra (Pryor & Collins 1996) was a second conditional POCL planner contemporaneous with CNLP. It took a slightly different tack, more in line with our category of POC planning. Rather than making non-deterministic actions primary, as did CNLP, Cassandra took initial-state uncertainty as primary, and modeled non-deterministic actions as actions with effects conditional on unknown (and unknowable) propositions. Plinth (Goldman & Boddy 1994a) was another early conditional planner, differing from Cassandra and CNLP in being a linear (total-order) planner; POCL planners were the dominant paradigm at that point. This made it considerably simpler, and it avoided the incompleteness issues of CNLP.

SADL (Golden & Weld 1996) is a language for planning and sensing that is based on ADL (Pednault 1989). The effects of its actions include both changes to the world state and information-gathering operations; the latter assign values to run-time variables. It uses a three-valued logic, with T, F, and U (for unknown). This is necessary because SADL, like the early conditional planners, does not have a representation of world state distinguished from the agent's knowledge of it. SADL supports universally quantified information goals, observational effects somewhat similar to ours, conditional effects that are activated via secondary preconditions, and universally quantified effects. Although SADL shares several characteristics with our formalism, we did not use it directly because it included several features we did not need for our translation technique.

PUCCINI (Golden 1998) is a partial-order planner built on SADL. PUCCINI is a planning engine for a Unix softbot that has imperfect information about its environment. PUCCINI is not a pure planner: it interleaves planning and execution. The title of the paper, "Leap Before You Look", refers to how PUCCINI uses conditional effects to do information gathering. To see if a conditional effect's secondary precondition holds, it can execute the action and then check to see if the conditional effect occurred.

SGP, Sensory Graphplan (Weld, Anderson, & Smith 1998), extends the well known Graphplan algorithm (Blum & Furst 1997) in order to deal with uncertainty in the initial states of planning problems. SGP represents this uncertainty via the set of all possible initial states that the planner could be in. Planning proceeds with generating a planning graph for each initial state in the input problem. SGP uses sensing actions in order to establish Graphplan-like mutex conditions among the possible planning graphs, which in turn, enables the planner to determine which planning graph it is in during the execution. Thus in a sense, SGP generates a conditional plan of the form a policy like CondFCP, represented by the set of planning graphs that the planner generates and the mutex condition between those graphs. In our experiments, we have not considered SGP because an examination of the experimental results in (Weld, Anderson, & Smith 1998) has revealed that both of the planning systems that we have used in our experiments in this paper, PKS (Petrick & Bacchus 2002; 2004) and MBP (Bertoli *et al.* 2001a), would easily outperform SGP in our experimental domains.

PKS is a simple forward-chaining planning algorithm capable of performing depth-first or breadth-first search over a space of "knowledge" states. Each knowledge state specifies a particular type of knowledge that the planner could use during planning, including facts known at planning time and facts whose truth value will be known at execution time. In this respect, PKS is different than most of the early planners described above. However, like many of them, PKS generates conditional plans which include branching points based on the knowledge for what is true/false at planning time and what will be true/false at execution.

MBP is probably the best-known planner for nondeterministic environments. It incorporates several algorithms based on symbolic model-checking (Cimatti *et al.* 2003; Cimatti, Roveri, & Traverso 1998; Daniele, Traverso, & Vardi 1999; Pistore, Bettin, & Traverso 2001). Solutions are classified as weak, strong, and strong-cyclic, and algorithms are provided for each. MBP has also been extended to deal with partial observability (Bertoli *et al.* 2001b; 2006) in nondeterministic domains. In these extensions, belief states are defined as a classes of states that represent common observations, and compactly represented using Binary Decision Diagrams (Bryant 1992).

Our work has been influenced by (Kuter & Nau 2004), which provided a way to take forward-chaining planners and modify them to run in fully observable nondeterministic planning domains. Our approach is analogous to theirs, in the sense that we have taken their FCP model of forward chaining planners and generalized it. However, our generalization and the formalism on which it is based go in a much different direction than theirs, since we have generalized in the direction of partially observable deterministic domains rather than fully observable nondeterministic domains.

Conclusions

We have presented a general technique for taking forward-chaining planners for deterministic domains and *conditionalizing* them, i.e., modifying them to produce conditional plans that solve partially observable planning problems.

In our experiments, we had expected CondSHOP2 (the conditionalized version of SHOP2) to outperform PKS (Petrick & Bacchus 2002; 2004) and MBP (Bertoli *et al.* 2001b); and indeed it did so—but it did so in a much more dramatic fashion than we had expected. CondSHOP2 easily solved all of the planning problems

in our experiments, but PKS and MBP were able to solve only the most trivial ones.

Like SHOP2, TLPlan and TALplanner are domain-configurable planners: their input includes domain-specific planning knowledge that it can make use of while solving planning problems. We have not implemented CondTLPlan and CondTALplanner, the conditionalized versions of TLPlan and TALplanner; but we have done an informal and approximate analysis of their complexity on the planning domains discussed here. Our analysis suggests that with appropriate control rules, CondTLPlan and CondTALplanner would perform similarly to CondSHOP2 in these domains.

The above observations suggest that the ability for a planner to use domain knowledge may be not only desirable, but indeed essential, to solve planning problems in partially observable domains. Our current understanding is that the reason is that such an ability might not only enable a planner to prune irrevelant actions, but also to prune irrelevant observations; i.e., it focuses the outcome plan's sensing to only those factors in the world that are relevant to the goals to be achieved by execution of the plan. We are currently working on an extensive theoretical and experimental study of our results to investigate this understanding more in detail. We are also considering Contingent-FF (Hoffmann & Brafman 2005) and POND (Bryce, Kambhampati, & Smith 2006) as part of this experimental study as recently, these planners have been demonstrated to be effective in comparison to MBP and PKS in some planning domains.

Our conditionalization technique is compatible with the *nondeterminization* technique described in (Kuter & Nau 2004), in the sense that both techniques can be combined, providing a way to translate classical planners into planners for domains that are both partially observable and nondeterministic. We have developed most of the theory for this translation, but have not yet done an implementation and experiments. We intend to do so in the near future.

Acknowledgments

This work was supported in part by DARPA's Transfer Learning and Integrated Learning programs, NSF grant IIS0412812, and AFOSR grants FA95500510298, FA95500610405, and FA95500610295. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001a. MBP: a model based planner. In *IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information*.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001b. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI-01*, 473–478. Seattle, USA: Morgan Kaufmann.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2006. Strong Planning under Partial Observability. *Artificial Intelligence* 170:337–384.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Planning Graph Heuristics for Belief Space Search. *Journal of Artificial Intelligence Research* 26:35–99.

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

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Strong planning in non-deterministic domains via model checking. In *AIPS-98*, 36–43. AAAI Press.

Daniele, M.; Traverso, P.; and Vardi, M. 1999. Strong cyclic planning revisited. In *ECP-99*, 35–48.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *KR-96*. 174–185.

Golden, K. 1998. Leap before you look: Information gathering in the PUCCINI planner. In *AIPS-98*, 70–77.

Goldman, R. P., and Boddy, M. S. 1994a. Conditional linear planning. In *AIPS-94*, 80–85.

Goldman, R. P., and Boddy, M. S. 1994b. Representing uncertainty in simple planners. In *KR-94*.

Hoffmann, J., and Brafman, R. 2005. Contingent Planning via Heuristic Forward Search with Implicit Belief States. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*.

Kuter, U., and Nau, D. 2004. Forward-chaining planning in nondeterministic domains. In *AAAI-2004*.

Kvarnström, J., and Doherty, P. 2001. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Articial Intelligence* 30:119–169.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

Nilsson, N. 1980. *Principles of Artificial Intelligence*. Morgan Kaufmann.

Pednault, E. P. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *KR-89*, 324–332.

Peot, M., and Smith, D. 1992. Conditional nonlinear planning. In *AIPS-92*, 189–197.

Petrick, R., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *AIPS-02*.

Petrick, R., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *ICAPS-04*, 2–11.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic techniques for planning with extended goals in non-deterministic domains. In *ECP-01*.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *JAIR* 4:287–339.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending Graphplan to handle uncertainty and sensing actions. In *AAAI-98*, 897–904.