Planning and Acting in Incomplete Domains

Paper #114

Abstract

Engineering complete planning domain descriptions is often very costly because of human error or lack of domain knowledge. Learning complete domain descriptions is also very challenging because many features are irrelevant to achieving the goals and data may be scarce. We present an agent that plans and acts in incomplete domains by i) synthesizing plans to avoid execution failure due to ignorance of the domain model, and ii) passively learning about the domain model during execution to improve later re-planning attempts.

Our planner `DeFAULT` is the first to reason about a domain's incompleteness to avoid potential plan failure. `DeFAULT` computes failure explanations for each action and state in the plan and counts the number of interpretations of the incomplete domain where failure will occur. We show that, depending on the domain, `DeFAULT` performs best by counting propositional models or prime implicants (diagnoses). Our agent `Goalie` learns about the preconditions and effects of incompletely-specified actions while monitoring its state and, in conjunction with `DeFAULT` plan failure explanations, can diagnose past and future action failures. We show that by reasoning about incompleteness in planning (as opposed to ignoring it) `Goalie` re-plans less, executes fewer actions, and achieves its goals faster.

Introduction

The knowledge engineering required to create complete and correct domain descriptions for planning problems is often very costly and difficult [**??**]. Machine learning techniques have been applied with some success [**?**], but still suffer from impoverished data and limitations of the algorithms [**?**]. In particular, we are motivated by applications in instructable computing [**?**] where a domain expert teaches an intelligent system about a domain, but can often leave out whole procedures (plans) and aspects of action descriptions. In such cases, the alternative to making domains complete, is to plan around the incompleteness. That is, given knowledge of the possible action descriptions, we seek out plans that will succeed despite any (or most) incompleteness in the domain formulation.

While prior work [**?**] (henceforth abbreviated, GL) has categorized risks to a plan and described plan quality metrics in terms of the risks (essentially single-fault diagnoses of plan failure [**??**]), no prior work has sought to deliberately synthesize low-risk plans. **?** (henceforth abbreviated, CA) learn about incomplete domains by synthesizing plans compatible with their knowledge, but do not attempt to make their plans robust to incompleteness. CA synthesize plans that will achieve the goals under any interpretation of the incomplete domain, however in doing so the planner can pick the most convenient interpretation and be overly optimistic. Further, while operating in non-deterministic domains, they do not address learning action preconditions and assume that their environment will provide action failure signals. We focus on deterministic environments, but lift their assumptions by learning preconditions and not relying on the environment signaling failure.

Our planner `DeFAULT` builds upon GL to synthesize plans robust to incompleteness and our agent `Goalie` builds upon CA to use robust plans while learning about all types of incomplete action features and diagnosing plan failure. We rely on representing our knowledge of the incomplete domain and anticipated plan failure with propositional sentences that are manipulated during plan synthesis and execution. The advantage of reasoning about the incompleteness in this fashion is that we can potentially avoid plan failure (i.e., unsatisfied (sub)goals), minimize re-planning, and often reduce overall planning and execution time over approaches that do not reason about incompleteness, such as the planner used by CA.

This paper is organized as follows. The next section details Incomplete STRIPS, the language we use to describe incomplete domains. We follow with our approach to plan synthesis and search heuristics. We discuss alternatives to reasoning about failure explanations, including model counting and prime implicant counting. We describe our execution monitoring and replanning strategy, and then provide an empirical analysis, related work, and conclusion.

Background & Representation

Incomplete STRIPS minimally relaxes the classical STRIPS model to allow for possible preconditions and effects. In the following, we review the STRIPS model and present incomplete STRIPS.

STRIPS Planning Domains: A STRIPS [**?**] planning domain D defines the tuple (P, A, I, G), where: P is a set of propositions; A is a set of action descriptions; $I \subseteq P$ defines a set of initially true propositions; and $G \subseteq P$ defines the goal propositions. Each action $a \in A$ defines $\text{pre}(a) \subseteq P$, a set of preconditions, $\text{add}(a) \subseteq P$, a set of add effects, and $\text{del}(a) \subseteq P$, a set of delete effects A plan $\pi = (a_0, ..., a_{n-1})$ in D is a sequence of actions, which corresponds to a sequence of states $(s_0, ..., s_n)$, where $s_0 = I$, $\text{pre}(a_t) \subseteq s_t$ for $t = 0, ..., n-1$, $G \subseteq s_n$, and $s_{t+1} = s_t \backslash \text{del}(a_t) \cup \text{add}(a_t)$ for $t = 0, ..., n-1$.

Incomplete STRIPS Domains: Incomplete STRIPS domains are identical to STRIPS domains, with the exception that the actions are incompletely specified. Much like planning with incomplete state information [**??**], the action incompleteness is not completely unbounded. The preconditions and effects of each action can be any subset of the propositions P; the incompleteness is with regard to a lack of knowledge about which of the subsets correspond to each precondition and effect. To narrow the possibilities, we find it convenient to refer to the *known*, *possible*, and *impossible* preconditions and effects. For example, an action's preconditions must consist of the known preconditions, and it must not contain the impossible preconditions, but we do not know if it contains the possible preconditions. The union of the known, possible, and impossible preconditions must equal P; therefore, an action can represent any two, and we can infer the third. We choose to represent the known and possible, but note that GL represent the known and impossible; with the trade-off making our representation more appropriate if there are fewer possible action features.

An incomplete STRIPS domain $\tilde{D}$ defines the tuple $(P, \tilde{A}, I, G)$, where: P is a set of propositions; $\tilde{A}$ is a set of incomplete action descriptions; $I \subseteq P$ defines a set of initially true propositions; and $G \subseteq P$ defines the goal propositions. Each action $\tilde{a} \in \tilde{A}$ defines $\text{pre}(\tilde{a}) \subseteq P$, a set of known preconditions, $\widetilde{\text{pre}}(\tilde{a}) \subseteq P$, a set of possible preconditions, $\text{add}(\tilde{a}) \subseteq P$, a set of known add effects, $\widetilde{\text{add}}(\tilde{a}) \subseteq P$, a set of possible add effects, $\text{del}(\tilde{a}) \subseteq P$, a set of known delete effects, and $\widetilde{\text{del}}(\tilde{a}) \subseteq P$, a set of possible delete effects

Consider the following incomplete domain: $P = \{p, q, r, g\}$, $\tilde{A} = \{\tilde{a}, \tilde{b}, \tilde{c}\}$, $I = \{p, q\}$, and $G = \{g\}$. The actions are defined: $\text{pre}(\tilde{a}) = \{p, q\}, \widetilde{\text{pre}}(\tilde{a}) = \{r\}, \widetilde{\text{add}}(\tilde{a}) = \{r\}, \widetilde{\text{del}}(\tilde{a}) = \{p\}$, $\text{pre}(\tilde{b}) = \{p\}, \text{add}(\tilde{b}) = \{r\}, \text{del}(\tilde{b}) = \{p\}, \widetilde{\text{del}}(\tilde{a}) = \{q\}$, and $\text{pre}(\tilde{c}) = \{r\}, \widetilde{\text{pre}}(\tilde{c}) = \{q\}, \text{add}(\tilde{c}) = \{g\}$.

The set of incomplete domain features $\mathcal{F}(\tilde{D})$ is comprised of the following propositions for each $\tilde{a} \in \tilde{A}$: $\widetilde{\text{pre}}(\tilde{a}, p)$ if $p \in \widetilde{\text{pre}}(\tilde{a})$, $\widetilde{\text{add}}(\tilde{a}, p)$ if $p \in \widetilde{\text{add}}(\tilde{a})$, and $\widetilde{\text{del}}(\tilde{a}, p)$ if $p \in \widetilde{\text{del}}(\tilde{a})$. An interpretation $\mathcal{F}^i(\tilde{D}) \subseteq \mathcal{F}(\tilde{D})$ of the incomplete STRIPS domain defines a STRIPS domain, in that every feature $f \in \mathcal{F}^i(\tilde{D})$ indicates that a possible precondition or effect is a respective known precondition or known effect; those features not in $\mathcal{F}^i(\tilde{D})$ are impossible preconditions or effects.

A plan π for $\tilde{D}$ is a sequence of actions, that when applied, *can lead* to a state where the goal is satisfied. A plan $\pi = (\tilde{a}_0, ..., \tilde{a}_{n-1})$ in an incomplete domain $\tilde{D}$ is sequence of actions, that corresponds to the *optimistic* sequence of states $(s_0, ..., s_n)$, where $s_0 = I$, $\text{pre}(\tilde{a}_t) \subseteq s_t$ for $t = 0, ..., n$, $G \subseteq s_n$, and $s_{t+1} = s_t \backslash \text{del}(\tilde{a}_t) \cup \text{add}(\tilde{a}_t) \cup \widetilde{\text{add}}(\tilde{a}_t)$ for $t = 0, ..., n-1$.

For example, the plan $(\tilde{a}, \tilde{b}, \tilde{c})$ corresponds to the state sequence $(s_0 = \{p, q\}, s_1 = \{p, q, r\}, s_2 = \{q, r\}, s_3 = \{q, r, g\})$, where the goal is satisfied in s_3.

Our definition of the plan semantics sets a loose requirement that plans with incomplete actions succeed under the most *optimistic* interpretation $\mathcal{F}^0(\tilde{D}) = \{\widetilde{\text{add}}(\tilde{a}, p) | \widetilde{\text{add}}(\tilde{a}, p) \in \mathcal{F}(\tilde{D})\}$ (i.e., possible preconditions need not be satisfied and the possible add effects (but not the possible delete effects) are assumed to occur when computing successor states). However, we would prefer that plans succeed under as many interpretations as possible; as we show, constructing plans that succeed for more interpretations decreases the number of instances where we must re-plan as we learn about the true domain.

Planning in Incomplete Domains

We present a forward state space planner called `DeFAULT` that attempts to minimize the number of interpretations of the incomplete domain that can result in plan failure. `DeFAULT` generates states reached under the optimistic interpretation of the incomplete domain, but labels each state proposition with the interpretations where it will be impossible to achieve the proposition. As such, the number of interpretations labeling the goals reached by a plan indicates the number of failed interpretations. By counting interpretations (i.e., propositional model counting), we can determine the quality (robustness) of a plan.

`DeFAULT` labels propositions and actions with domain interpretations that will respectively fail to achieve the proposition or fail to achieve the preconditions of an action. That is, labels indicate the cases where a proposition will be false (i.e., the plan fails to establish the proposition). Labels $d(\cdot)$ are represented as propositional sentences over $\mathcal{F}(\tilde{D})$ whose models correspond to domain interpretations.

Initially, each proposition $p_0 \in s_0$ is labeled $d(p_0) = \bot$ to denote that there are no failed interpretations affecting the initial state, and each $p_0 \notin s_0$ is labeled $d(p_0) = \top$. For all $t \geq 0$, we define:

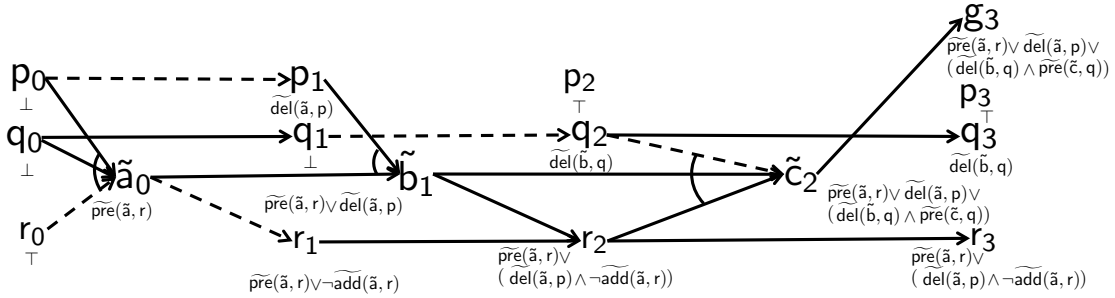

Figure 1: Labeled Plan

$$d(\tilde{a}_t) = d(\tilde{a}_{t-1}) \vee \bigvee_{p \in \mathrm{pre}(\tilde{a})} d(p_t) \vee \bigvee_{p \in \widetilde{\mathrm{pre}}(\tilde{a})} (d(p_t) \wedge \widetilde{\mathrm{pre}}(\tilde{a}_t, p)) \quad (1)$$

$$d(p_{t+1}) = \begin{cases} d(p_t) \wedge d(\tilde{a}_t) & : p \in \mathrm{add}(\tilde{a}_t) \\ d(p_t) \wedge (d(\tilde{a}_t) \vee & \\ \quad \widetilde{\neg \mathrm{add}}(\tilde{a}_t, p)) & : p \in \widetilde{\mathrm{add}}(\tilde{a}_t) \\ \top & : p \in \mathrm{del}(\tilde{a}_t) \\ d(p_t) \vee \widetilde{\mathrm{del}}(\tilde{a}_t, p) & : p \in \widetilde{\mathrm{del}}(\tilde{a}_t) \\ d(p_t) & : \text{otherwise} \end{cases} \quad (2)$$

where $d(\tilde{a}_{-1}) = \bot$. The intuition behind the label propagation is that in Equation 1 an action will fail in the domain interpretations $d(\tilde{a}_t)$ where a prior action failed, a known precondition is not satisfied, or a possible precondition (which is a known precondition for the interpretation) is not satisfied. As defined by Equation 2, the plan will fail to achieve a proposition at time $t+1$ in all interpretations where i) the plan fails to achieve the proposition at time t and the action fails, ii) the plan fails to achieve the proposition at time t and the action fails or it does not add the proposition in the interpretation, iii) the action deletes the proposition, iv) the plan fails to achieve the proposition at time t or in the interpretation the action deletes the proposition, or v) the action does not affect the proposition and any prior failed interpretations still apply.

A consequence of our definition of action failure is that each action fails if any prior action fails. This definition follows from the semantics that the state becomes undefined if we apply an action whose preconditions are not satisfied. While we use this notion in plan synthesis, we explore the semantics that the state does not change (i.e., it is defined) upon failure when we discuss acting in incomplete domains. The reason that we define action failures in this manner is that we can determine all failed interpretations affecting a plan $d(\pi)$, defined by $d(\tilde{a}_{n-1}) \vee \bigvee_{g \in G} d(g_n)$. It is possible to determine the interpretations that fail to successfully execute the plan up to and including time t by computing $d(\tilde{a}_t)$.

For example, consider the plan depicted in Figure 1. The propositions in each state and each action at each time are labeled by the propositional sentence below it. The edges in the figure connecting the propositions and actions denote what must be true to successfully execute an action or achieve a proposition. The dashed edges indicate that action incompleteness affects the ability of an action or proposition to support a proposition. For example, $\tilde{a}$ possibly deletes p, so the edge denoting its persistence is dashed. The propositional sentences $d(\cdot)$ below each proposition and action denote the domain interpretations where a action will fail or a proposition will not be achieved. For example, $\tilde{b}$ at time one, $\tilde{\mathsf{b}}_1$, will fail if either $\mathrm{pre}(\tilde{a}, r)$ or $\mathrm{del}(\tilde{a}, p)$ is true in the interpretation. Thus, $d(\pi) = \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \vee (\widetilde{\mathrm{del}}(\tilde{b}, q) \wedge \widetilde{\mathrm{pre}}(\tilde{c}, q))$ and any domain interpretation satisfying $d(\pi)$ will fail to execute the plan and achieve the goal.

Heuristics In Incomplete Domains

Similar to propagating failed interpretation labels in a plan, we can propagate labels in the relaxed planning problem to compute a search heuristic. The heuristic is the number of actions in a relaxed plan, and, while we do not use the number of failed domain interpretations as the primary heuristic, we use the failure labels to bias the selection of the relaxed plan actions and break ties between search nodes with an equivalent number of actions in their relaxed plans. We solve the relaxed planning problem using a planning graph and we start with a brief description of planning graphs in STRIPS domains.

Planning Graph Heuristics: A relaxed planning graph is a layered graph of sets of vertices $(\mathcal{P}_t, \mathcal{A}_t, ..., \mathcal{A}_{t+m}, \mathcal{P}_{t+m+1})$. The planning graph built for a state s_t defines $\mathcal{P}_t = \{p_t | p \in s_t\}$, $\mathcal{A}_{t+k} = \{a_t | \forall_{p \in \mathrm{pre}(a)} p_t \in \mathcal{P}_{t+k}, a \in A \cup A(P)\}$, and $\mathcal{P}_{t+k+1} = \{p_{t+k+1} | a_{t+k} \in \mathcal{A}_{t+k}, p \in \mathrm{add}(a)\}$, for $k = 0, ..., m$. The set $A(P)$ includes noop actions for each proposition, such that $A(P) = \{a(p) | p \in P, \mathrm{pre}(a(p)) = \mathrm{add}(a(p)) = p, \mathrm{del}(a(p)) = \emptyset\}$. The h^{FF} heuristic [**?**] solves this relaxed planning problem by choosing actions from $\mathcal{A}_{t+m}$ to support the goals in $\mathcal{P}_{t+m+1}$, and recursively for each chosen action's preconditions, counting the number of chosen actions.

Incomplete Domain Heuristics: Propagating failed interpretations in the planning graph resembles propagating failed interpretations over a plan. The primary difference is how we define the failed interpretations for a proposition when the proposition has multiple sources

of support; recall that we allow only serial plans and at each time each state proposition is supported by persistence and/or a single action – action choice is handled in the search space. In a level of the relaxed planning graph, there are potentially many actions supporting a proposition, and we select the supporter with the fewest failed interpretations. The chosen supporting action, denoted $\hat{a}_{t+k}(p)$, determines the failed interpretations affecting a proposition p at level $t+k+1$.

A relaxed planning graph with propagated labels is a layered graph of sets of vertices of the form $(\hat{\mathcal{P}}_t, \hat{\mathcal{A}}_t, ..., \hat{\mathcal{A}}_{t+m}, \hat{\mathcal{P}}_{t+m+1})$. The relaxed planning graph built for a state $\tilde{s}_t$ defines $\hat{\mathcal{P}}_0 = \{\hat{p}_t | p \in \tilde{s}_t\}$, $\hat{\mathcal{A}}_{t+k} = \{\hat{a}_{t+k} | \forall_{p \in \text{pre}(\tilde{a})} \hat{p}_{t+k} \in \hat{\mathcal{P}}_{t+k}, \tilde{a} \in \tilde{A} \cup A(P)\}$, and $\hat{\mathcal{P}}_{t+k+1} = \{p_{t+k+1} | \hat{a}_{t+k} \in \hat{\mathcal{A}}_{t+k}, p \in \text{add}(\tilde{a}) \cup \widetilde{\text{add}}(\tilde{a})\}$, for $k = 0, ..., m$. Much like the successor function used to compute next states, the relaxed planning graph assumes an optimistic semantics for action effects by adding possible add effects to proposition layers, but, as we will explain below, it associates failed interpretations with the possible adds.

Each planning graph vertex has a label, denoted $\hat{d}(\cdot)$. The failed interpretations $\hat{d}(p_t)$ affecting a proposition are defined such that $\hat{d}(p_t) = d(p_t)$, and for $k \geq 0$,

$$\widehat{d}(\tilde{a}_{t+k}) = \bigvee_{p \in \text{pre}(\tilde{a})} \hat{d}(p_{t+k}) \vee \bigvee_{p \in \widetilde{\text{pre}}(\tilde{a})} (\hat{d}(p_{t+k}) \wedge \widetilde{\text{pre}}(\tilde{a}, p)) \quad (3)$$

$$\hat{d}(p_{t+k+1}) = \begin{cases} \widehat{d}(\hat{a}_{t+k}(p)) & : p \in \text{add}(\hat{a}_{t+k}(p)) \\ \widehat{d}(\hat{a}_{t+k}(p)) \vee & \\ \neg\widetilde{\text{add}}(\hat{a}_{t+k}(p), p) & : p \in \widetilde{\text{add}}(\hat{a}_{t+k}(p)) \end{cases} \quad (4)$$

Every action in every level k of the planning graph will fail in any interpretation where their preconditions are not supported (Equation 3). A proposition will fail to be achieved in any interpretation where the chosen supporting action fails to add the proposition (Equation 4).

We note that the rules for propagating labels in the planning graph differ from the rules for propagating labels in the state space. In the state space, the action failure labels include interpretations where any prior action fails. In the relaxed planning problem, the action failure labels include only interpretations affecting the action's preconditions, and not prior actions; it is not clear which actions will be executed prior to achieving a proposition because many actions may be used to achieve other propositions at the same time step.

Heuristic Computation: We terminate the relaxed planning graph expansion at the level $t+k+1$ where one of the following conditions is met: i) the planning graph reaches a fixed point where the labels do not change, $\hat{d}(p_{t+k}) = \hat{d}(p_{t+k+1})$ for all $p \in P$, or ii) the goals have been reached at $t+k+1$ and the fixed point has not yet been reached. Our $h^{\sim FF}$ heuristic makes use of the chosen supporting action $\hat{a}_{t+k}(p)$ for each proposition that requires support in the relaxed plan, and, hence, measures the number of actions used while attempting to minimize failed interpretations. The other heuristic $h^{\sim M}$ measures the number of interpretations that fail to reach the goals in the last level (i.e., such that $h^{\sim M} = |M(\vee_{g \in G} \hat{d}(g_{t+m+1}))|$, where $m+1$ is the last level of the planning graph and $M(\psi)$ is the set of models of a propositional sentence ψ. `DeFAULT` uses both heuristics, treating $h^{\sim FF}$ as the primary heuristic and using $h^{\sim M}$ to break ties.

Counting Models and Prime Implicants

Failure explanations $d(\cdot)$ and $\hat{d}(\cdot)$ are propositional sentences that help bias decisions in search and heuristics. Namely, we assume that we can count the number of propositional models of these sentences to indicate how many interpretations of the incomplete domain will fail to successfully execute a plan. Model counting is intractable [**?**], but by representing the sentences as OBDDs [**?**], model counting is polynomial in the size of the OBDD [**?**] (which can be exponential sized in the worst case).

In addition to OBDDs and model counting, we also explore counting prime implicants (PIs) – also called diagnoses. A set of PIs is a set of conjunctive clauses (similar to a DNF) where no clause is subsumed by another, and are used in model-based diagnosis to represent diagnoses (sets of incomplete features that must interact to cause system failure) [**??**]. We find it useful to bound the cardinality (the number of conjuncts) of the PIs, effectively over-approximating the models of a propositional sentence.

Instead of counting the models of two labels $d(\cdot)$ and $d'(\cdot)$, we can compare the number of PIs. Our intuition is that having fewer diagnoses of failure is preferred, just as is having fewer models of failure (even though having fewer PIs does not always imply fewer models). The advantage is that counting PIs is much less expensive than counting models, especially if we bound the cardinality of the PIs. Finally, we use a heuristic when counting PIs whereby we compare two sets in terms of the number of cardinality-one PIs, and if equal, the number of cardinality-two PIs, and so on. The intuition behind comparing PIs in this fashion is that smaller PIs are typically satisfied by a larger number of models and are thus more representative. That is, a sentence with one cardinality-one PI will have more models than a sentence with one cardinality-two PI.

Acting in Incomplete Domains

Acting in incomplete domains provides an opportunity to learn about the domain by observing the states resulting from action application. In the following, we describe what our agent `Goalie` can learn from acting in incomplete domains and how it might achieve its goals. `Goalie` will continue to execute a plan until it is faced with an action that is guaranteed to fail or it has determined that the plan failed in hindsight.

`Goalie` maintains a propositional sentence ϕ defined over $\mathcal{F}(\tilde{D}) \cup \{fail\}$ which describes the current knowl-

edge of the incomplete domain. The proposition $fail$ denotes whether `Goalie` believes that its current plan failed – it is not always possible to determine if an action applied in the past did not have its preconditions satisfied. Initially, `Goalie` believes $\phi = \top$, denoting its complete lack of knowledge of the incomplete domain and whether its current plan will fail. If `Goalie` executes an action a in state s and transitions to state s', then it can update its knowledge ϕ as $\phi \wedge o(s, a, s')$, where

$$o(s,a,s') = \begin{cases} (fail \wedge o^-) \vee o^+ & : s = s' \\ o^+ & : s \neq s' \end{cases} \quad (5)$$

$$o^- = \bigvee_{\substack{\widetilde{\text{pre}}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): \\ p \notin s}} \widetilde{\text{pre}}(\tilde{a}, p) \quad (6)$$

$$o^+ = o^{pre} \wedge o^{add} \wedge o^{del} \quad (7)$$

$$o^{pre} = \bigwedge_{\substack{\widetilde{\text{pre}}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): \\ p \notin s}} \neg\widetilde{\text{pre}}(\tilde{a}, p) \quad (8)$$

$$o^{add} = \bigwedge_{\substack{\widetilde{\text{add}}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): \\ p \in s' \setminus s}} \widetilde{\text{add}}(\tilde{a}, p) \wedge \bigwedge_{\substack{\widetilde{\text{add}}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): \\ p \notin s \cup s'}} \neg\widetilde{\text{add}}(\tilde{a}, p) \quad (9)$$

$$o^{del} = \bigwedge_{\substack{\widetilde{\text{del}}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): \\ p \in s \setminus s'}} \widetilde{\text{del}}(\tilde{a}, p) \wedge \bigwedge_{\substack{\widetilde{\text{del}}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): \\ p \in s \cap s'}} \neg\widetilde{\text{del}}(\tilde{a}, p) \quad (10)$$

We assume that the state will remain unchanged when `Goalie` executes an action whose precondition is not satisfied by the state, and because the state is observable, Equation 5 references the case where the state does not change and the case where it changes. If the state does not change, then either the action failed and one of its unsatisfied possible preconditions is a precondition (Equation 6) or the action succeeded (Equation 7). If the state changes, then `Goalie` knows that the action succeeded. If an action succeeds, `Goalie` can conclude that i) each possible precondition that was not satisfied is not a precondition (Equation 8), ii) each possible add effect that appears in the successor but not the predecessor state is an add effect and each that does not appear in either state is not an add effect, iii) each possible delete effect that appears in the predecessor but not the successor is a delete effect and each that appears in both states is not a delete effect.

Using ϕ, it is possible to determine if the next action in a plan, or any subsequent action, can or will fail. If $\phi \wedge d(a_{t+k})$ is satisfiable, then a_{t+k} *can* fail, and if $\phi \models d(a_{t+k})$, then a_{t+k} *will* fail. `Goalie` will execute an action if it may not fail, even if later actions in its plan will fail. If `Goalie` determines that its next action will fail, or a prior action failed ($\phi \models fail$), then it will re-plan. `Goalie` uses ϕ to modify the actions during re-planning by checking for each incomplete domain feature $f \in \mathcal{F}(\tilde{D})$ if $\phi \models f$ or if $\phi \models \neg f$. Each such literal entailed by ϕ indicates if the respective action has the possible feature as a known or impossible feature; all other features remain as possible features.

Algorithm 1: `Goalie`$(s, G, \tilde{A})$

Input: state s, goal G, actions $\tilde{A}$

1 $\phi \leftarrow \top$; $\pi \leftarrow Plan(s, G, \tilde{A}, \phi)$;
2 **while** $\pi \neq ()$ *and* $G \not\subseteq s$ **do**
3 $\quad a \leftarrow \pi.first()$; $\pi \leftarrow \pi.rest()$;
4 $\quad$ **if** $pre(a) \subseteq s$ *and* $\phi \not\models \bigvee_{\widetilde{pre}(\tilde{a},p) \in \mathcal{F}(\tilde{D}): p \notin s} \widetilde{pre}(\tilde{a}, p)$ **then**
5 $\quad\quad s' \leftarrow Execute(a)$;
6 $\quad\quad \phi \leftarrow \phi \wedge o(s, a, s')$;
7 $\quad\quad s \leftarrow s'$;
8 $\quad$ **else**
9 $\quad\quad \phi \leftarrow \phi \wedge fail$;
10 $\quad$ **end**
11 $\quad$ **if** $\phi \models fail$ **then**
12 $\quad\quad \phi \leftarrow \exists_{fail}\phi$;
13 $\quad\quad \pi \leftarrow Plan(s, G, \tilde{A}, \phi)$;
14 $\quad$ **end**
15 **end**

Algorithm 1 is the strategy used by `Goalie`. The algorithm involves initializing the agent's knowledge and plan (line 1), and then while the plan is non-empty and the goal is not achieved (line 2) the agent proceeds as follows. The agent selects the next action in the plan (line 3) and determines if it can apply the action (line 4). If it applies the action, then the next state is returned by the environment/simulator (line 5) and the agent updates its knowledge (line 6 and Equation 5) and state (line 7), otherwise the agent determines that the plan will fail (line 9). If the plan has failed (line 11), then the agent forgets its knowledge of the plan failure (line 12) and finds a new plan using its new knowledge (line 13). `Goalie` is not guaranteed success, unless it can find a plan that will not fail (i.e., $d(\pi) = \perp$).

`Goalie` is not hesitant to apply actions that may fail because trying actions is its only way to learn about them. However, `Goalie` is able to determine when actions will fail and re-plans. More conservative strategies are possible if we assume that `Goalie` can query a knowledge engineer about action features to avoid potential plan failure, but we leave such goal-directed knowledge acquisition for future work.

Revising Existing Plans

I'll use the notion of failure explanations on causal links to talk about plans. A few observations:

* A valid plan will have all of its (sub)goals supported by causal links that are free of failure explanations. * A subgoal that is supported by a causal link with no failure explanation (a known link) does not require any other causal links. * An action that does not support required causal links is not required. * A subgoal is not required if it supports an action that is not required. * A causal link is not required if it supports a subgoal that is not required.

We have two types of constructs actions and causal links, and failure explanations (a propositional sentence) and a required flag are associated with each. A causal link c is a triple (a, p, a') denoting that a provides p for a'. A failure explantation $d(\cdot)$ describes interpretations of an incomplete domain that will fail to achieve an action or causal link. A required flag is denoted $r(\cdot)$.

We can compute the causal links of a plan $(a_{-1}, a_0, ..., a_n)$, as follows. Define sets C and $E_{-1}, ..., E_{n-1}$ that are the respective causal links and potential causal links at each time. Initialize E_{-1} with elements $e = (a_{-1}, p, \emptyset)$ for each $p \in \text{add}(a_{-1})$, where $d(e) = \bot$ and $r(e) = \bot$,. For each next action a_i in the plan, we add to C any causal links (a, p, a_i) that arise in supporting the preconditions of a_i so that

$C = C \cup \{c = (a, p, a_i) | e = (a, p, \cdot) \in E_{i-1}, p \in \text{pre}(a_i) \cup \widetilde{\text{pre}}(a_i), d(c) = d(e)\}$

We define E_i, $0 \leq i < n$, as a set of new potential causal links generated or possibly threatened by a_i so that

$$
\begin{aligned}
E_i = &\{e = (a_j, p, \emptyset) | e \in E_{i-1}, p \notin \text{del}(a_i), j < i\} \cup \\
&\{e = (a_i, p, \emptyset) | p \in \text{add}(a_i), d(e) = d(a_i)\} \cup \\
&\{e = (a_i, p, \emptyset) | p \in \widetilde{\text{add}}(a_i), d(e) = d(a_i) \vee \neg\widetilde{\text{add}}(a_i, p)\} \cup \\
&\{e = (a_i, p, \emptyset) | p \in \widetilde{\text{del}}(a_i), d(e) = d(a_i) \vee \widetilde{\text{del}}(a_i, p)\}
\end{aligned}
$$

Upon generating the set of causal links C, we can determine which are required to achieve the goals. We assert that the goal action is required $r(a_n) = \top$, thereby forcing us to consider which of the causal links supporting it are required or not. If there is an action a_i that is required, then we must achieve each of its known preconditions (we may or may not achieve its possible preconditions). For each $p \in \text{pre}(a_i)$ there exists a set of causal links $C(p, i) = \{c = (a, p, a_i) | c \in C\}$; if the exists a $c \in C(p, i)$ where $d(c) = \bot$, then for all other $c' \in C(p, i)$ we set $r(c') = \bot$ and set $r(c) = \top$.

Empirical Evaluation

The empirical evaluation is divided into four sections: the domains used for the experiments, the test setup used, results for off-line planning, and results for on-line planning and execution. The questions that we would like to answer include:

- Q1: Does reasoning about incompleteness lead to high quality plans?
- Q2: Does counting prime implicants perform better than counting models ?
- Q3: Does reasoning about incompleteness save overall time in planning and execution?

Domains: There are four domains that we use in the evaluation: a modified Pathways, Bridges, a modified PARC Printer, and Barter World. In Pathways, Bridges, and Barter World, we derived multiple instances by randomly (with probabilities 0.25, 0.5, 0.75, and 1.0 for each action) injecting incomplete domain features. In PARC Printer we injected incomplete domain features with probability 0.5. With these variations of the domains, the instances include up to ten thousand incomplete domain features each, with many of the domains using several such features in each action instance. All off-line planning results are taken from ten random instances (varying $\tilde{A}$) of each problem (same I and G). All on-line planning and execution results are taken from ten randomly selected interpretations (ground-truth domains used by the execution simulator) of the ten instances per problem used in off-line planning. The problem instances and generators are available at *withheld for blind review.*

The Pathways domain from the International Planning Competition (IPC) [**?**] involves actions that model chemical reactions in signal transduction pathways. Pathways is a naturally incomplete domain where the lack of knowledge of the reactions is quite common because they are an active research topic in biology. We introduced each type of incompleteness to model incomplete knowledge of products required, created, or destroyed by reactions.

The Bridges domain consists of a traversable grid and the task is to find a different treasure at each corner of the grid. There are three versions where each subsequent version has an additional type of incompleteness. In Bridges1, a bridge might be required to cross between some grid locations (a possible precondition). In Bridges2, many of the bridges may have a troll living underneath that will take all the treasure accumulated (a possible delete effect). In Bridges3, the corners may give additional treasures (possible add effects). Grids are square and vary in dimension (2, 4, 8, and 16).

The PARC Printer domain from the IPC involves planning paths for sheets of paper through a modular printer. A source of domain incompleteness is that a module accepts only certain paper sizes, but its documentation is incomplete. Thus, paper size becomes a possible precondition to actions using the module.

The Barter World domain involves navigating a grid and bartering items to travel between locations. Items are available at different locations and may be required to travel between other locations. The domain is incomplete because some of the actions that acquire certain items are not always known to be successful (possible add effects) and traveling between locations may require certain items (possible preconditions) and may result in the loss of an item (possible delete effects). The instances involve different size grids and numbers of items. Grids vary in dimension (2, 4, 8, and 16).

Test Setup: The tests were run on a Linux machine with a 3 Ghz Xeon processor, a memory limit of 2GB, and a time limit of 20 minutes per run for the off-line planning invocation and 60 minutes for each on-line planning and execution invocation. All code was written in Java and run on the 1.6 JVM. `DeFAULT` uses a greedy best first search with deferred heuristic evaluation and a dual-queue for preferred and non-preferred operators [**?**].

We use five configurations of the planner: DeFAULT-FF, DeFAULT-PIk (k = 1, 2, 3), and DeFAULT-BDD, that differ in how they reason about domain incompleteness. DeFAULT-FF does not compute failure explanations and uses the FF heuristic; it is inspired by the planner used by CA because it is likely to find a plan that will work for only the most optimistic domain interpretation. DeFAULT-PIk, where k is the bound on the cardinality of the prime implicants, uses only prime implicants to compare failure explanations. DeFAULT-BDD uses OBDDs to represent and count failure explanations. The number of failed interpretations for a plan π found by any of the planners is reported herein by counting models of an OBDD representing $d(\pi)$. The versions of the planner are compared by the proportion of interpretations of the incomplete domain that achieve the goal and total planning time in seconds. The plots in the following section depict these results using the cumulative percentage of successful domain interpretations and planning time to identify the performance over all problems and domains. We also report detailed results on the number of solved problems per domain.

Off-line Planning Results: Figure 2 includes two plots that illustrate the cumulative proportion of domain interpretations that will successfully execute each plan for each problem (top) and cumulative total planning time (bottom). To enhance readability, every two hundredth data point is plotted in the figures (while still representative of the true cumulative number). Table 1 lists the number of solved problems for each planner and highlights the most solved for each domain in bold.

We see that Q1 is answered mostly positively by the results. Plan quality is improved by reasoning about incompleteness (through DeFAULT-PIk or -BDD), but scalability suffers. However, we note that minimizing the number of failed interpretations of a domain can be phrased as a conformant probabilistic planning problem, which is notoriously difficult [**??**], and expecting the scalability of a classical planner is perhaps unreasonable.

Q2 is answered overall negatively by our experiments because the BDD approach solves more problems with better quality and in less time than the PIk approaches. However, we note that $PI1$ performs best in the Barter World domain, which was specifically designed to have failure explanations with high cardinality prime implicants, which also has an impact upon OBDD-based representations. In this domain, we are faced with a high cost for model counting that becomes prohibitive. Thus, counting bounded prime implicants is a viable option when model counting is too costly.

On-line Planning and Execution Results: Figure 3 depicts a comparison between Goalie using DeFAULT-FF and DeFAULT-BDD to synthesize plans, so that we can judge whether planning and execution strategies such as that of CA will benefit from planners reason about incompleteness. The scatter plots in the figure show the respective number of actions applied to achieve the goal, the number of plans generated (initial

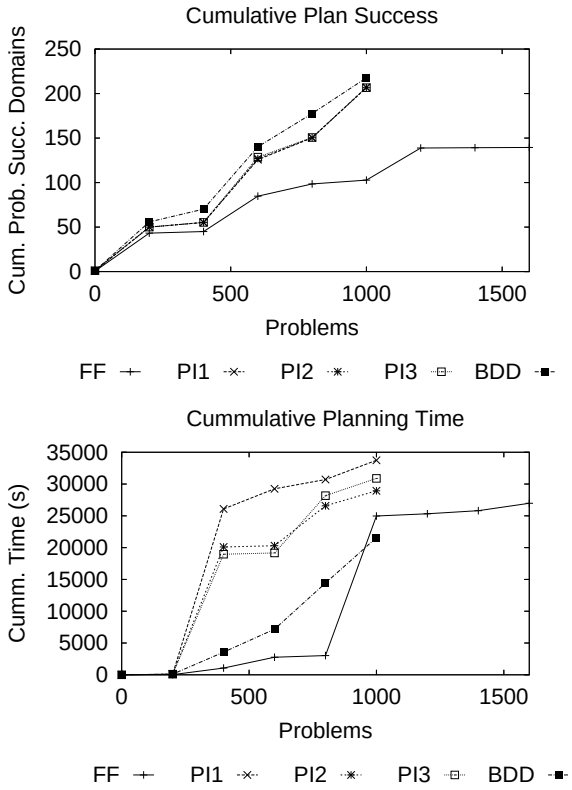

Figure 2: Cumulative Plan Success and Planning Time

plan plus re-planning episodes), the total planning and execution time, and the ratio of the number of domain interpretations after learning in an episode to the number before. The total time plot includes points plotted at one hour (the time limit) for instances where with only one of the planners Goalie achieves the goal.

Q3 is answered positively, but only in the most challenging instances. The plot of the total time taken shows that the planners are somewhat mixed or even for times less than 100 seconds. However, for times greater than 100 seconds, it appears that using DeFAULT-BDD in Goalie can take up to an order of magnitude less time. By investigating the plots of the number of actions taken and the number of plans generated, it is apparent why DeFAULT-BDD might take less overall time: reasoning about incompleteness leads to less replanning and overall fewer steps to achieve the goals.

Another interesting observation is that the fourth plot in Figure 3 indicates that upon goal achievement DeFAULT-BDD typically eliminates fewer of the possible interpretations of incomplete domain than DeFAULT-FF. The explanation for this outcome is that by failing less, there is less opportunity to learn from failures, and by taking fewer actions overall, there are fewer actions that can be learned about.

Related Work

Planning in incomplete domains is noticeably similar to planning with incomplete information, where action de-

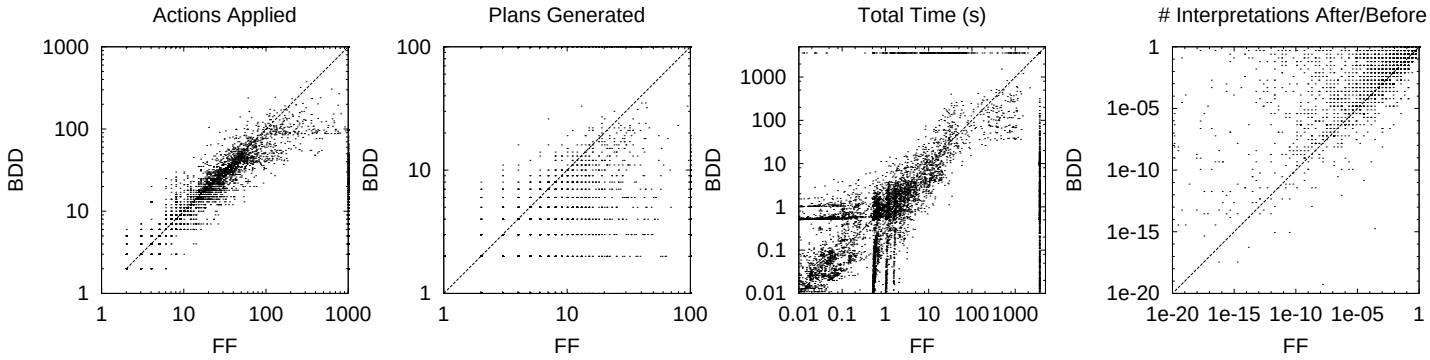

Figure 3: Comparison of Goalie with DeFAULT-FF and DeFAULT-BDD.

Domain	FF	PI1	PI2	PI3	BDD
PARC Printer	169	97	97	100	**140**
Bridges1 0.25	40	**40**	39	39	**40**
Bridges1 0.5	40	35	34	34	**40**
Bridges1 0.75	40	31	30	30	**32**
Bridges1 1.0	40	**30**	20	20	**30**
Bridges2 0.25	40	39	39	39	**40**
Bridges2 0.5	40	34	35	33	**40**
Bridges2 0.75	40	**31**	29	30	30
Bridges2 1.0	40	**30**	29	28	**30**
Bridges3 0.25	40	**40**	**40**	**40**	**40**
Bridges3 0.5	40	**40**	**40**	**40**	**40**
Bridges3 0.75	40	38	38	38	**40**
Bridges3 1.0	40	**37**	**37**	36	36
Bridges Total	480	425	410	407	**438**
Barter 0.25	120	**120**	**120**	**120**	**120**
Barter 0.5	120	**120**	**120**	**120**	117
Barter 0.75	120	**120**	119	**120**	111
Barter 1.0	12	**12**	**12**	**12**	11
Barter Total	372	**372**	371	**372**	359
Pathways 0.25	140	50	40	40	**60**
Pathways 0.5	140	**70**	60	50	60
Pathways 0.75	150	**60**	40	40	**60**
Pathways 1.0	170	50	60	60	**70**
Pathways Total	600	230	200	190	**250**
Total	1621	1124	1078	1069	**1187**

Table 1: Instances Solved By Domain

scriptions instead of states are incomplete. Incomplete domains can be translated to conformant probabilistic planning domains, and planners such as POND [**?**] and PFF [**?**] are applicable. However, while the translation is theoretically feasible, practical issues regarding numeric precision prohibit effective use of existing planners. While we do not report results, we translated the instances described here and ran both POND and PFF; both planners compared equally with DeFAULT on the problems where numeric precision was not exceeded.

Our investigation is an instantiation of model-lite planning [**?**]. Constraint-based hierarchical task networks are an alternative, pointed out by **?**, which avoid specifying all preconditions and effects through methods and constraints that correspond to underlying, implicit causal links.

As previously stated, this work is a natural extension of the **?** model for evaluating plans in incomplete domains. Our methods for computing plan failure explanations are slightly different in that we compute them in the forward direction and allow for multiple, interacting faults instead of the single faults. In addition to calculating the failure explanations of partial plans, we have also presented a relaxed planning heuristic informed by failure explanations.

Prior work of **?** addresses planning with incomplete models, but does not attempt to synthesize robust plans, which is similar to our DeFAULT-FF planner. We have shown that incorporating knowledge about domain incompleteness into the planner can lead to a more effective agent. We also differ in that we do not assume direct feedback from the environment about action failures and we can learn action preconditions.

Conclusion

We have presented the first work to address planning in incomplete domains as heuristic search to find robust plans. Our planner, DeFAULT, i) performs forward search while maintaining plan failure explanations, and ii) estimates the future failures by propagating failure explanations on planning graphs. We have shown that, compared to a planner that essentially ignores aspects

of the incomplete domain, DeFAULT is able to scale reasonably well but find much better quality plans. We have also shown that representing plan failure explanations with prime implicants leads to better scalability in complex domains, but by counting OBDDs models performs better in more domains. Our agent Goalie is most effective when using DeFAULT, can learn incomplete actions, and can diagnose future and past failures.