Planning and Acting in Incomplete Domains

Christopher Weber and **Daniel Bryce**
christopherweber@hotmail.com, daniel.bryce@usu.edu
Department of Computer Science
Utah State University

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 a planner and agent that respectively plan and act 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 `DeFault` performs best by counting prime implicants (failure diagnoses) rather than propositional models. 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 (as opposed to ignoring it) `Goalie` fails and re-plans less and executes fewer actions.

1 Introduction

The knowledge engineering required to create complete and correct domain descriptions for planning problems is often very costly and difficult (Kambhampati 2007; Wu, Yang, and Jiang 2007). Machine learning techniques have been applied with some success (Wu, Yang, and Jiang 2007), but still suffer from impoverished data and limitations of the algorithms (Kambhampati 2007). In particular, we are motivated by applications in instructable computing (Mailler et al. 2009) 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 of the) incompleteness in the domain formulation.

While prior work (Garland and Lesh 2002) (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 (de Kleer and Williams 1987)), no prior work has sought to deliberately synthesize low-risk plans based on incomplete STRIPS-style domains (notable work in Markov decision processes (Nilim and El Ghaoui 2005; Choudhary et al. 2006) and model-based reinforcement learning (Sutton and Barto 1998) has explored similar issues). Our planner `DeFault` labels partial plans with propositional explanations of failure due to incompleteness (derived from the semantics of assumption-based truth maintenance systems (Bryce 2011)) and either counts failure models or prime implicants (diagnoses) to bias search. Our agent `Goalie` passively learns about the incomplete domain as it executes actions, like Chang and Amir (2006) (henceforth abbreviated, CA). Unlike CA, `Goalie` executes plans that are robust to domain incompleteness. Within `Goalie`, we compare the use of robust plans generated by our planner `DeFault`, and plans that are generated in the spirit of CA which are not intentionally robust (i.e., they are optimistically successful). We demonstrate that the effort to synthesize robust plans is justified because `DeFault` executes fewer overall actions and fails and re-plans less.

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 re-planning strategy, and then provide an empirical analysis, related work, and conclusion.

2 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 Domains: A STRIPS (Fikes and Nilsson 1971) 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 (Domshlak and Hoffmann 2007; Bryce, Kambhampati, and Smith 2008), 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, noting that 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:
$P = \{p, q, r, g\}, \tilde{A} = \{\tilde{a}, \tilde{b}, \tilde{c}\}, I = \{p, q\}, G = \{g\}$
The actions are defined:
$\text{pre}(\tilde{a}) = \{p, q\}, \widetilde{\text{pre}}(\tilde{a}) = \{r\}, \widetilde{\text{add}}(\tilde{a}) = \{r\}, \widetilde{\text{del}}(\tilde{a}) = \{p\}$
$\text{pre}(\tilde{b}) = \{p\}, \text{add}(\tilde{b}) = \{r\}, \text{del}(\tilde{b}) = \{p\}, \widetilde{\text{del}}(\tilde{b}) = \{q\}$
$\text{pre}(\tilde{c}) = \{r\}, \widetilde{\text{pre}}(\tilde{c}) = \{q\}, \text{add}(\tilde{c}) = \{g\}$

The set of incomplete domain features F 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 $\mathsf{F}^i \subseteq \mathsf{F}$ of the incomplete STRIPS domain defines a STRIPS domain, in that every feature $f \in \mathsf{F}^i$ indicates that a possible precondition or effect is a respective known precondition or known effect. Those features not in F^i are not 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 a 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.

Discussion: Our definition of the plan semantics sets a loose requirement that plans with incomplete actions succeed under the most *optimistic* conditions: 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. This notion of optimism is similar to that of GraphPlan (Blum and Furst 1995) in that both assert every proposition that could be made true at a particular time even if only a subset of the propositions can actually be made true. In GraphPlan, there *may* exist a plan to establish a proposition if the proposition appears in the planning graph. In our definitions there *does* exist an interpretation of the incomplete domain that will establish a proposition if it appears in a state (Bryce 2011), and this interpretation *may* correspond to the true domain. In GraphPlan, failing to assert a proposition that may be established could eliminate plans, and in our case, failing to assert a proposition would prevent us from computing interpretations of the incomplete domain that achieve the goal.

We ensure that the plan is valid for the least constraining (most optimistic) interpretation of the incomplete domain. If the plan can achieve the goal in the most optimistic interpretation, then it may achieve the goal in others, if the goal is not reachable in this interpretation, then it cannot be reached in any interpretation (Bryce 2011). As we will show, we can efficiently determine the interpretations in which a plan is invalid and use the number of such failed interpretations as a plan quality metric.

3 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 (a failure explanation) 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 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 F 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

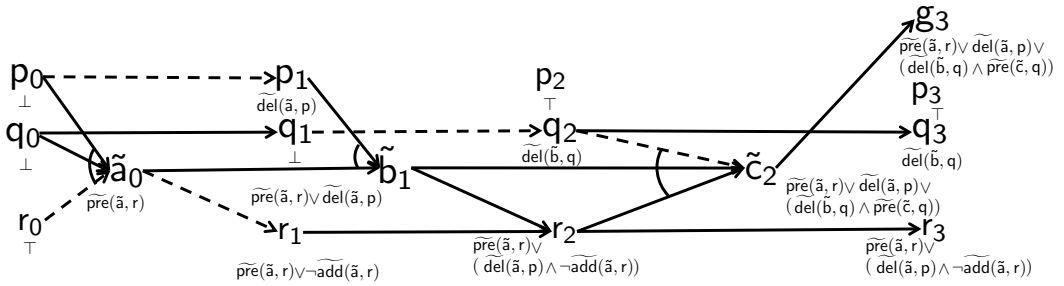

Figure 1: Labeled Plan

$t \geq 0$, we define:

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

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

where $d(\tilde{a}_{-1}) = \perp$. 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 $d(\tilde{a}_{n-1}) \vee \bigvee_{g \in G} d(g_n)$. By $d(\tilde{a}_t)$, it is possible to determine the interpretations that fail to successfully execute the plan up to and including time 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, $\tilde{b}_1$, will fail if either $\widetilde{\text{pre}}(\tilde{a}, r)$ or $\widetilde{\text{del}}(\tilde{a}, p)$ is true in the interpretation. Thus, $d(\pi) = \widetilde{\text{pre}}(\tilde{a}, r) \vee \widetilde{\text{del}}(\tilde{a}, p) \vee (\widetilde{\text{del}}(\tilde{b}, q) \wedge \widetilde{\text{pre}}(\tilde{c}, q))$ and any domain interpretation satisfying $d(\pi)$ will fail to execute the plan and achieve the goal.

4 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 primary heuristic is the number of actions in a relaxed plan. 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. As in recent trends in satisficing planning (classical, conformant, etc.) we want high quality solutions, but not at the expense of returning no solution. We solve the relaxed planning problem using a planning graph and thus begin with a brief description of planning graphs.

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 $\mathcal{P}_t = \{p_t | p \in s_t\}$, $\mathcal{A}_{t+k} = \{a_t | \forall_{p \in \text{pre}(a)} p_t \in \mathcal{P}_{t+k}, a \in A \cup A(P)\}$, and $\mathcal{P}_{t+k+1} = \{p_{t+k+1} | a_{t+k} \in \mathcal{A}_{t+k}, p \in \text{add}(a)\}$, for $k = 0, ..., m$. The set $A(P)$ includes noop actions for each proposition, such that $A(P) = \{a(p) | p \in P, \text{pre}(a(p)) = \text{add}(a(p)) = p, \text{del}(a(p)) = \emptyset\}$. The h^{FF} heuristic (Hoffmann and Nebel 2001) 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 that 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 $\hat{\mathcal{P}}_{t+k+1} = \{\hat{p}_{t+k+1} | \hat{a}_{t+k} \in \hat{\mathcal{A}}_{t+k}, p \in \text{add}(\tilde{a}) \cup \widetilde{\text{add}}(\tilde{a})\}$, for $k = 0, ..., m$. Much like the successor function used to compute next states, the relaxed planning graph assumes an optimistic semantics for effects by adding possible add effects to proposition layers. However, 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 $\hat{d}(\hat{p}_t)$ affecting a proposition are defined such that: $\hat{d}(\hat{p}_t) = d(p_t)$; and for $k \geq 0$,

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

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

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, an action's failure labels include only the interpretations affecting its 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.

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 explanations do not change, $\hat{d}(\hat{p}_{t+k}) = \hat{d}(\hat{p}_{t+k+1})$ 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. Hence, it measures the number of actions used while attempting to minimize failed interpretations (the supporting actions are chosen by comparing failure explanations). Our $h^{\sim M}$ heuristic measures the number of interpretations that fail to reach the goals in the last level: $h^{\sim M} = |M(\vee_{p \in G} \hat{d}(\hat{p}_{t+m+1}))|$, where $m+1$ is the last level of the planning graph, $M(\cdot)$ is the set of models of a formula. `DeFault` uses both heuristics, treating $h^{\sim FF}$ as the primary heuristic and using $h^{\sim M}$ to break ties. While it is likely that swapping the role of the heuristics may lead to higher quality plans (fewer failed interpretations), our informal experiments determined that the scalability of `DeFault` is greatly limited in such cases – measuring failed interpretations is not correlated with solution depth in the search graph, unlike relaxed plan length. The relaxed plans are informed by the propagated explanations because we use the failure explanation to bias action selection.

5 Counting Models & Prime Implicants

Failure explanations $d(\cdot)$ and $\hat{d}(\cdot)$ are propositional sentences that help bias decisions in our heuristic-based search. 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 (Roth 1996), but by representing the sentences as OBDDs (Bryant 1986), model counting is polynomial in the size of the OBDD (Darwiche and Marquis 2002), although it 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) which allows us to computer a heuristic $h^{\sim PI}$, which is similar to $h^{\sim M}$. A set of PIs is a set of conjunctive clauses – similar to a DNF, where no clause is subsumed by another. These are used in model-based diagnosis to represent diagnoses – sets of incomplete features that must interact to cause system failure (de Kleer and Williams 1987). 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 (as in $h^{\sim PI}$). 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, when counting PIs, we use a heuristic that compares 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 of the number of models. That is, a sentence with one cardinality-one PI will have more models than a sentence with one cardinality-two PI.

6 Acting in Incomplete Domains

Acting in incomplete domains provides an opportunity to learn about the domain by observing the states resulting from execution. In the following, we describe what our agent `Goalie` can learn from acting in incomplete domains and how it achieves 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 $\mathsf{F} \cup \{fail\}$, which describes the current knowledge of the incomplete domain. The proposition $fail$ denotes

whether Goalie believes that its current plan may have 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 $\tilde{a}$ in state s and transitions to state s', then it updates its knowledge as $\phi \wedge o(s, \tilde{a}, s')$, where

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

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

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

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

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

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

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 both 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 (Equation 9); 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 (Equation 10).

Using ϕ, it is possible to determine if the next action in a plan, or any subsequent action, can or will fail. If $\phi \wedge d(\tilde{a}_{t+k})$ is satisfiable, then $\tilde{a}_{t+k}$ *can* fail, and if $\phi \models d(\tilde{a}_{t+k})$, then $\tilde{a}_{t+k}$ *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 $f \in \mathsf{F}$ if $\phi \models f$ or if $\phi \models \neg f$. Each such literal entailed by ϕ indicates that the respective action has the possible feature as a known or impossible feature; all other features remain as possible features.

Algorithm 1 is the strategy used by Goalie. First, the algorithm initializes the agent's knowledge and plan (line 1). Then while the plan is non-empty and the goal is not achieved (line 2) the agent proceeds with execution. The

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

```
   Input: state s, goal G, actions Ã
 1 φ ← ⊤; π ← Plan(s, G, Ã, φ);
 2 while π ≠ () and G ⊈ s do
 3 |  ã ← π.first(); π ← π.rest();
 4 |  if pre(ã) ⊆ s and φ ⊭ ⋁_{pre(ã,p)∈F: p∉s} pre(ã, p) then
 5 |  |  s' ← Execute(ã);
 6 |  |  φ ← φ ∧ o(s, ã, s');
 7 |  |  s ← s';
 8 |  else
 9 |  |  φ ← φ ∧ fail;
10 |  end
11 |  if φ ⊨ fail then
12 |  |  φ ← ∃_{fail} φ;
13 |  |  π ← Plan(s, G, Ã, φ);
14 |  end
15 end
```

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 cannot 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 an action will fail and so re-plans. More conservative strategies are possible if we assume that Goalie can query a domain expert about action features to avoid potential plan failure, but we leave such goal-directed knowledge acquisition for future work.

7 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 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: As the number of incomplete features grows, does stronger reasoning about incompleteness help?
- Q4: Does reasoning about incompleteness reduce the number of execution failures during execution?

Domains: We use four domains in the evaluation: a modified Pathways, Bridges, a modified PARC Printer, and Barter World. For these domains, we created multiple instances by injecting incomplete features with probabilities

0.25, 0.5, 0.75, and 1.0. An instance may possess up to ten thousand incomplete features. Planning results are taken from ten random instances (varying F) of each problem. Within these, each planning and execution result is one of ten ground-truth domains selected by the simulator. The problem generators and our planner are available at: http://www.cs.usu.edu/∼danbryce/software/default.jar.

The Pathways (PW) 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, and are an active research topic in biology (Choudhary et al. 2006).

The Bridges (BR) domain, of which there are three versions, consists of a traversable grid where the task is to find a different treasure at each corner of the grid. In BR1 (version 1), a bridge might be required to cross between some grid locations (a possible precondition); in BR2, many of the bridges may have a troll living underneath that will take all the treasure accumulated (a possible delete effect); and in BR3, the corners may give additional treasures (possible add effects). Grids are square and vary in dimension (2-16).

The PARC Printer (PP) 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, in using the module, paper size becomes a possible precondition to actions.

The Barter World (BW) domain involves navigating a grid and bartering items to travel between locations. The domain is incomplete because actions that acquire items are not always known to be successful (possible add effects) and traveling between locations may both require certain items (possible preconditions) and result in the loss of an item (possible delete effects). Grids vary in dimension (2-16) and items in number (1-4).

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 test. All code is written in Java and run on the 1.6 JVM.

We use five configurations of the planner: DeFault-FF, DeFault-PIk (k = 1, 2, 3), and DeFault-BDD, each of which differ in how they reason about domain incompleteness. DeFault-FF does not compute failure explanations and uses the FF heuristic; inspired by the planner used by CA, 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, counts prime implicants to compare failure explanations. DeFault-BDD uses OBDDs to represent and count failure explanations. DeFault uses a best first search with deferred heuristic evaluation and a dual-queue for preferred and non-preferred operators (Helmert 2006).

The number of failed interpretations for a plan π found by any of the planners is found 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

	FF	PI1	PI2	PI3	BDD
FF	0	155	161	161	123
PI1	**629**	0	**79**	**78**	**208**
PI2	**619**	77	0	46	**208**
PI3	**594**	62	**51**	0	**199**
BDD	**512**	189	189	187	0

Table 1: Number of plans having a greater number of successful domain interpretations (i.e., better quality). Bold indicates best performers.

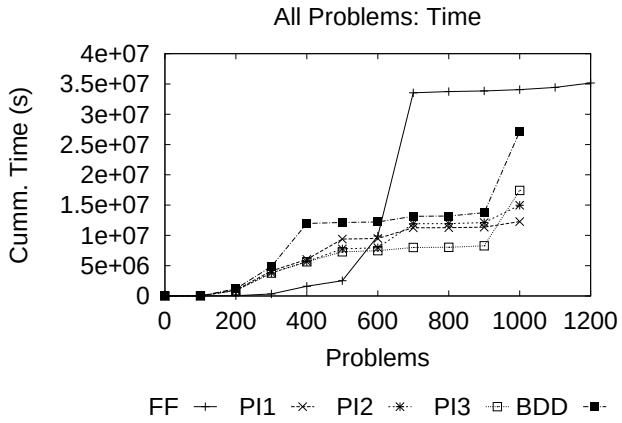

Figure 2: Cumulative Time in All Domains.

in seconds. The plot in the following section depict these results using the cumulative planning time to identify the performance over all problems and domains. We also report detailed results on the number of solved problems per domain and the relative quality of solutions (successful domain interpretations).

We also compare the off-line planning results to a conformant probabilistic planner POND (with N=10 particles in its heuristic) (Bryce, Kambhampati, and Smith 2008) that solves translated instances of the incomplete STRIPS problems. We set the minimum required probability of goal satisfaction to the minimum proportion of successful domain interpretations of plans found by the other approaches. We do not provide the details of the translation because the results are very poor, but refer the reader to an extended version of this work (Bryce 2011). We attempted a comparison to PFF (Domshlak and Hoffmann 2007), but the implementation proved unstable for all but the smallest instances.

Off-line Planning Results: Figure 2 plots the cumulative total planning time. To enhance readability, every one hundredth data point is plotted in the figures (while still representative of the true cumulative number). Table 1 lists the number of times that each configuration finds a better solution (number of successful interpretations) than another; for example, $PI1$ finds a better solution than FF 629 times. Table 2 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 positively by the results. Plan quality is improved by reasoning about incompleteness

Domain	FF	PI1	PI2	PI3	BDD	POND
PP 0.25	130	83	85	**86**	80	10
PP 0.5	130	87	**88**	87	80	0
PP 0.75	130	82	**83**	81	80	0
PP 1.0	13	**10**	9	9	8	0
PP	403	262	**265**	263	248	10
BR1 0.25	40	**22**	**22**	**22**	**22**	2
BR1 0.5	39	**20**	**20**	**20**	**20**	2
BR1 0.75	36	**19**	**19**	**19**	**19**	2
BR1 1.0	4	**2**	**2**	**2**	**2**	1
BR2 0.25	38	20	20	20	**21**	3
BR2 0.5	35	**25**	**25**	**25**	23	3
BR2 0.75	35	**22**	21	21	21	2
BR2 1.0	4	**2**	**2**	**2**	**2**	1
BR3 0.25	45	**36**	**36**	**36**	**36**	1
BR3 0.5	47	**33**	**33**	**33**	32	2
BR3 0.75	46	39	39	39	**41**	1
BR3 1.0	5	**4**	**4**	**4**	3	1
BR	374	**244**	243	243	242	21
BW 0.25	150	106	128	**129**	108	60
BW 0.5	150	134	**137**	134	118	45
BW 0.75	150	**140**	138	137	111	27
BW 1.0	15	**14**	**14**	**14**	11	2
BW	465	394	**417**	414	348	155
PW 0.25	160	**40**	**40**	**40**	**40**	19
PW 0.5	160	**70**	60	50	60	13
PW 0.75	170	**60**	50	40	**60**	12
PW 1.0	19	5	6	6	**7**	2
PW	509	**175**	156	136	167	46
Total	1751	1075	**1081**	1056	1005	232

Table 2: Instances Solved By Domain

(through DeFault-PIk or -BDD), but scalability suffers. However, we note that minimizing the number of failed interpretations can be phrased as a conformant probabilistic planning problem, which is notoriously difficult (Domshlak and Hoffmann 2007; Bryce, Kambhampati, and Smith 2008), and expecting the scalability of a classical planner is perhaps unreasonable.

Q2 is answered overall positively by our experiments because the PIk counting approaches solve more problems with better quality and in less time than the BDD model counting approach.

Q3 is answered negatively because as the probability of injecting incomplete features grows from 0.25 to 1.0 the $PI3$ approach initially solves the most problems, but then $PI2$, and then $PI1$ solve the most problems in each domain. A possible explanation for this result is that it becomes too costly to reason about incompleteness as it increases and that a more coarse approach is needed; however, the BDD approach, while not the best, seems to degrade less as the incompleteness increases. It is likely that the OBDD package implementation (Vahidi 2011) is to credit for the BDD approach's performance because model counting can become prohibitively expensive in larger problems.

Table 2 indicates that POND is not competitive and suggests that existing approaches are not directly applicable to planning in incomplete domains. We note that DeFault is inspired by POND, but employs more approximate reasoning about incompleteness by using bounded prime implicants (see (Bryce 2011) for a more thorough discussion).

On-line Planning & Execution Results: Figure 3 depicts a comparison between Goalie using DeFault-FF and DeFault-$PI1$ to synthesize plans, so that we can judge whether planning and execution strategies such as that of CA will benefit when 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, and the total planning and execution time.

Q4 is answered mostly positively. By investigating the plots of the number of actions taken and the number of plans generated, it is apparent that DeFault-$PI1$ takes fewer actions as the instances require near 100 steps, and tends to fail and re-plan less often. The plot of the total time taken shows that the planners are somewhat mixed or even for times less than 10 seconds. However, for times greater than 10 seconds, it appears that using DeFault-FF in Goalie can take up to an order of magnitude less time. However, there are several difficult instances in which DeFault-$PI1$ does outperform DeFault-FF. We expect that more efficient implementations of reasoning about prime implicants (e.g., tries) could lower the cost of planning with DeFault, allowing it to capitalize on its more robust plans.

8 Related Work

Planning in incomplete domains is noticeably similar to planning with incomplete information. However, for incomplete domains it is the actions, not the states, that are incomplete. Incomplete domains can be translated to conformant probabilistic planning domains, and planners such as POND (Bryce, Kambhampati, and Smith 2008) and PFF (Domshlak and Hoffmann 2007) are applicable. However, while the translation is theoretically feasible, current CPP planners are not reasonable approaches to the problem (Bryce 2011).

Our investigation is an instantiation of model-lite planning (Kambhampati 2007). Constraint-based hierarchical task networks are an alternative, pointed out by Kambhampati (2007), 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 Garland and Lesh (2002) model for evaluating plans in incomplete domains. We note that their STRIPS-like formulation of incomplete domains has come to define the term "incomplete domains" as a research area. Our methods for computing plan failure explanations are different in that we compute them in the forward direction and allow for multiple, interacting faults instead of single faults. In addition to calculating the failure explanations of partial plans, we use a relaxed planning heuristic informed by failure explanations.

Prior work of Chang and Amir (2006) 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

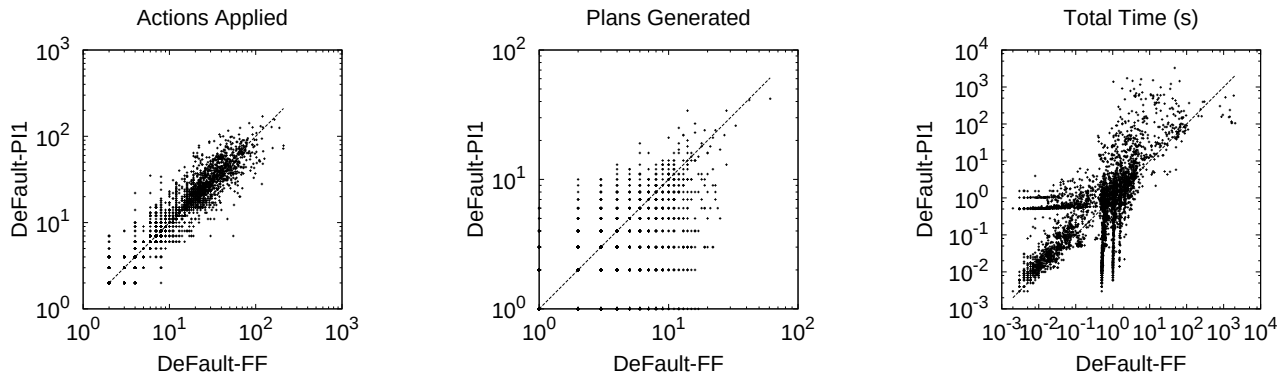

Figure 3: Comparison of Goalie with DeFault-FF and DeFault-$PI1$.

incompleteness into the planner can lead to an agent that replans less and thus fails less often. We also differ in that we do not assume direct feedback from the environment about action failures and we can learn action preconditions.

9 Conclusion

We have presented the first work to utilize heuristic search to find robust plans for incomplete domains. 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 configuration 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 than counting OBDDs models. Our agent Goalie uses DeFault to generate robust plans that fail less often and require less re-planning than more optimistic planning approaches that ignore incompleteness.

Acknowledgements: This work was supported by DARPA contract HR001-07-C-0060.

References

Blum, A., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proceedings of IJCAI'95*, 1636–1642.

Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677–691.

Bryce, D.; Kambhampati, S.; and Smith, D. 2008. Sequential monte carlo in probabilistic planning reachability heuristics. *AIJ* 172(6-7):685–715.

Bryce, D. 2011. Planning in incomplete domains. Technical Report 001, Utah State University. Available at: http://www.cs.usu.edu/∼danbryce/papers/USU-CS-TR-11-001.pdf.

Chang, A., and Amir, E. 2006. Goal achievement in partially known, partially observable domains. In *Proceedings of ICAPS'06*.

Choudhary, A.; Datta, A.; Bittner, M. L.; and Dougherty, E. R. 2006. Intervention in a family of boolean networks. *Bioinformatics* 22(2):226–232.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *JAIR* 17:229–264.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *AIJ* 32(1):97–130.

Domshlak, C., and Hoffmann, J. 2007. Probabilistic planning via heuristic forward search and weighted model counting. *JAIR* 30:565–620.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of AAAI'71*, 608–620.

Garland, A., and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. In *Proceedings of AAAI'02*.

Helmert, M. 2006. The fast downward planning system. *JAIR* 26:191–246.

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

Kambhampati, S. 2007. Model-lite planning for the web age masses. In *Proceedings of AAAI'07*.

Mailler, R.; Bryce, D.; Shen, J.; and Orielly, C. 2009. Mable: A framework for natural instruction. In *Proceedings of AAMAS'09*.

Nilim, A., and El Ghaoui, L. 2005. Robust control of Markov decision processes with uncertain transition matrices. *Oper. Res.* 53(5):780–798.

Roth, D. 1996. On the hardness of approximate reasoning. *AIJ* 82(1-2):273–302.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

Vahidi, A. 2011. JDD: Java BDD package. http://javaddlib.sourceforge.net/jdd/.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *K. Eng. Rev.* 22(2):135–152.