View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@USU

Utah State University

DigitalCommons@USU

All Graduate Theses and Dissertations

Graduate Studies

5-2012

Planning in Incomplete Domains

Jared William Robertson

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Robertson, Jared William, "Planning in Incomplete Domains" (2012). *All Graduate Theses and Dissertations*. 1435.
https://digitalcommons.usu.edu/etd/1435

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

UtahStateUniversity MERRILL-CAZIER LIBRARY

PLANNING IN INCOMPLETE DOMAINS

by

Jared Robertson

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Daniel Bryce Major Professor	Dr. Nicholas Flann Committee Member
Dr. Vicki H. Allan Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2012

Copyright © Jared Robertson 2012
All Rights Reserved

ABSTRACT

Planning in Incomplete Domains

by

Jared Robertson, Master of Science

Utah State University, 2012

Major Professor: Dr. Daniel Bryce
Department: Computer Science

Engineering complete planning domain descriptions is often very costly because of human error or lack of domain knowledge. While many have studied knowledge acquisition, relatively few have studied the synthesis of plans when the domain model is incomplete (i.e., actions have incomplete preconditions or effects). Prior work has evaluated the correctness of plans synthesized by disregarding such incomplete features, but not how to synthesize plans by reasoning about the incompleteness. In this work, we describe several techniques for reasoning that takes into account action incompleteness to increase the number of interpretations under which the plans will succeed. Among the techniques, we show that representing explanations of plan failure with prime implicants provides a natural approach to comparing plans by counting prime implicants instead of models – leading to better scalability and comparable quality plans.

We present and empirically evaluate a forward heuristic search planner, called `DeFAULT`, that synthesizes plans by propagating information about faults due to incompleteness both within the state space and the relaxed planning space. We compare `DeFAULT` with a control planner that uses the fast forward (FF) heuristic (measuring plan length and ignoring incompleteness). The results show that `DeFAULT` i) scales comparable to the planner using the FF heuristic (while finding better solutions), and ii) scales better when counting prime implicants than models.

(71 pages)

PUBLIC ABSTRACT

Planning in Incomplete Domains

Automated planning in computer science consists of finding a sequence of actions leading from an initial state to a goal state. People who have expert knowledge of the specific problem domain work with experts in automated planning to define the domain states and actions. This knowledge engineering required to create complete and correct domain descriptions for planning problems is often very costly and difficult. Our goal with incomplete planning is to allow people to program domains without the need for planning experts.

Throughout the process of instruction of intelligent systems, teachers can often leave out whole procedures 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 incompleteness in the domain formulation.

A state in a domain consists of a set of propositions that can be either true or false. Actions in a domain require specific propositions to be true for the action to occur. Actions then add and remove propositions from the state to create a subsequent state. A valid plan consists of a sequence of actions that, starting with the initial state, change to match the goal state. An incomplete domain contains the same qualities as a complete domain, with the additional abilities of actions to possibly require a proposition to be true to initiate the action, as well as possibly adding and possibly removing propositions in the subsequent state. Actions that have possible preconditions and effects are referred to as incomplete actions.

Because no prior work exists for the purpose of empirical comparisons, we compare our incomplete action planner, which we call DeFAULT, with a traditional planner that assumes all good possibilities and no bad possibilities will occur. DeFAULT finds much better quality plans than the traditional planner while maintaining similar speed.

ACKNOWLEDGMENTS

I would like to thank the eternally patient Dr. Daniel Bryce for his encouragement and mentoring. I never would have learned as much as I have without your vast knowledge and guidance. I also thank my committee members for their willingness to participate in this process.

I give special thanks to my family, especially - specifically my father and my wife, for their encouragement, moral support, and patience.

This work was supported by the DARPA Bootstrapped Learning contract HR001-07-C-0060.

Jared Robertson

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, NOTATION, DEFINITIONS, ETC.

INTRODUCTION

The knowledge engineering required to create complete and correct domain descriptions for planning problems is often very costly and difficult [1, 2]. Machine learning techniques have been applied with some success [2, 3], but still suffer from impoverished data and limitations of the algorithms [1]. In particular, we are motivated by applications in instructable computing [4] wherein a domain expert teaches an intelligent system about a particular 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 [5] has categorized faults to a plan's correctness and described plan quality metrics in terms of the faults (essentially single-fault diagnoses of plan failure [6, 7]), no prior work has sought to deliberately synthesize low-fault plans. Specifically, the authors of [5] (henceforth abbreviated, GL) identify four types of plan faults: open preconditions (due to incomplete preconditions), possible clobberers (due to incomplete delete effects), unlisted effects (due to incomplete add effects), and false preconditions. GL develop an algorithm that steps backward through the plan to identify the "critical faults" – those instances wherein incomplete domain features can cause plan failure. For example, a possible clobber is a critical fault when (if it is truly a delete effect) it threatens a precondition or goal. The number of critical faults is an important measure of plan quality/correctness, that, unfortunately, no known planners seek to minimize (aside from our prior work [8] on single-fault planning, upon which this work is based).

Consider the following action that is taken from a modified version of the International Planning Competition (IPC) [9] PARC printer domain:

```
(:action HtmOverBlack-Move-A4
 :parameters ( ?sheet - sheet_t )
 :precondition (and (clear) (Available HtmOverBlack-RSRC)
                    (Sheetsize ?sheet A4)
                    (Location ?sheet HtmOverBlack_Entry-
                        EndCap_Exit))
 :effect (and (not (Available HtmOverBlack-RSRC))
              (Location ?sheet HtmOverBlack_Exit-
                  Down_TopEntry)
              (not (Location ?sheet HtmOverBlack_Entry-
                  EndCap_Exit))
              (Available HtmOverBlack-RSRC))
 :poss-effect (and (not (clear))))
```

The action models a modular printer component that prints on a sheet of A4-sized paper. The action is incomplete because it has a possible effect that the component will become jammed `(not (clear))`. The intuition behind the action is that the component manufacturer did not provide complete specifications, and it is unknown if feeding an A4 sheet will cause a paper jam. Note that an incomplete action is different from a non-deterministic action because each application of the incomplete action has the same effect at runtime; however, it is not clear what the effect will be at planning time. The action incompleteness can cause plan failure, as in the case of our example, by threatening the precondition of a later action (e.g., the precondition `(clear)` is threatened in a second application of the `HtmOverBlack-Move-A4` action).

Interpretations of Incompleteness

A pessimistic approach to reasoning about incomplete actions might assume that possible delete effects will always occur. Plans found under this pessimistic interpretation will be correct despite any action incompleteness, but are likely to be few or nonexistent. In the PARC printer example, a pessimistic interpretation will likely lead to proving that no plan exists, even though it is possible that the action does not have the delete effect on `(clear)`. Alternatively, an optimistic interpretation might assume that no possible delete effect occurs, in which case the planner can ignore that `(clear)` may be deleted. The optimistic interpretation is equally flawed because the action may actually delete `(clear)`. Instead, we adopt a cautiously optimistic interpretation wherein, like the optimistic interpretation, we assume that possible delete effects do not occur, but we also temper our optimism. We compute an explanation for cases under which each proposition that is optimistically true might be false. For example, after applying the action above, we would assert that `(clear)` is true, subject to the assumption that `(clear)` is not a delete effect of the action. Under these cautiously optimistic semantics, we can determine which interpretations of incomplete actions will result in failed goal achievement by inspecting the assumptions under which the goals are false. Plans that fail under fewer interpretations are preferred.

Failure Explanations and Counting

We take three qualitatively different approaches to recording a failure explanation for each proposition established at different times by a plan. The first, our control, amounts to the optimistic interpretation by recording no explanation for the failure to achieve a proposition. The second and third approaches represent failure explanations with propositional sentences, whose models correspond to interpretations of the incomplete actions. The second approach relies on intuitions from model-based diagnosis to represent each failure explanation by a set of diagnoses (each diagnosis is a conjunction of incomplete action features – i.e., a prime implicant). The third approach represents failure explanations by ordered binary decision diagrams (OBDDs). The second and third approaches provide a representation suitable for counting interpretations of the incomplete action features (i.e., propositional models) under which a proposition is achieved or not. The primary difference is that model counting with prime implicants is intractable [10], but polynomial in the size of an OBDD [11]. While we use each of the three approaches during plan synthesis to compare plans (in varying capacities), we use the third to provide a final assessment of a plan's quality: the number of interpretations of the incomplete actions under which the plan fails. That is, we describe several heuristic techniques to speed-up plan synthesis that are based on a particular representation of the failure explanations, but compare the resulting plans with a single, non-heuristic method.

For example, the first approach is entirely heuristic because it completely ignores failure explanations. In the second approach, we represent the failure explanations by prime implicants, and instead of counting models, we count the number of prime implicants. Counting prime implicants is a computationally inexpensive heuristic that assumes fewer diagnoses means fewer failed interpretations of the incomplete actions. The third method counts the actual number of failed action interpretations by representing them as an OBDD (which can be exponential-sized) and performing OBDD model counting (which is polynomial in the OBDD size). We claim that counting diagnoses (prime implicants) is more computationally feasible than counting OBDD models and the resulting plans are of similar quality, and that ignoring incompleteness altogether leads to poor quality plans.

Our claims are based upon GL's focus on counting a plan's critical risks as a measure of its quality. We observe that GL's definition of critical risks is equivalent to computing single-fault diagnoses, which allows us to generalize their notions to multi-fault diagnoses. Intuitively, the more diagnoses for plan failure, the fewer interpretations of the incomplete domain to achieve the goal. Naturally, a single-fault diagnosis covers more interpretations than a double- or triple-fault, so we count not just the number of diagnoses, but those of different cardinality. We stress that counting diagnoses is an approximation to counting models, but it nevertheless leads to more efficient planners that find comparable quality solutions.

Planners

We present a forward heuristic planner, called `DeFAULT`, that propagates failure explanations in the state space and relaxed planning problems. `DeFAULT` associates a set of explanations with each time step, i.e., each state in the search space or each planning graph layer in the relaxed planning problem. `DeFAULT`'s heuristic biases search toward plans that will fail in the fewest interpretations of the incomplete domain as possible. Because no prior work exists for the purpose of empirical comparisons, we not only compare DeFAULT with a planner that uses the FF heuristic and ignores domain incompleteness, but we also attempt a *more fair* comparison with a conformant probabilistic planner.

Our results indicate that DeFAULT can find much better quality plans than a planner that ignores incompleteness. In the following, we provide background on the representation of the planning problems studied, a discussion of languages used to capture incomplete actions, a formulation of failure explanations, a definition of diagnosis and model counting, a planner based on failure propagation, a relaxed planning heuristic for failure propagation, empirical evaluation, related work, and conclusion.

BACKGROUND AND REPRESENTATION

Planning consists of finding a sequence of actions that will achieve a specified goal. Classical planning deals with domains that are fully observable, deterministic, finite, static, and discrete. This work concerns itself with complete and incomplete planning models. In the following, we define each model, the related action representations, and plan semantics.

Complete Planning Domains

Complete planning domains correspond to the classical planning model, expressed using STRIPS actions [13]. STRIPS is a formal language for specifying planning problems.

Definition 1. *A complete planning domain D defines the tuple* $(\boldsymbol{P}, \boldsymbol{A}, \boldsymbol{a_{-1}}, \boldsymbol{a_n})$, *where*

- *P is a set of propositions - Boolean statements about the state of the domain*
- *A is a set of complete action descriptions, where each a □ A defines*
    - *pre*(a) ⊆ *P, a set of preconditions - propositions that must be true in order for the action to occur*
    - *add*(a) □ *P, a set of add effects - propositions set to true in the subsequent state*
    - *del*(a) □ *P, a set of delete effects - propositions set to false in the subsequent state*
- *add*(a_{-1}) □ *P defines a set of initially true propositions*

- *pre(a_n) □ P defines the goal propositions - propositions that must be true for the plan to succeed*

Example 1. For example, consider the following domain, which we will use as a running example:

- $P = \{p, q, r, g\}$
- $A = \{a, b, c\}$
  - $\mathrm{pre}(a) = \{p, q\}, \mathrm{add}(a) = \{r\}, \mathrm{del}(a) = \{\}$
  - $\mathrm{pre}(b) = \{p\}, \mathrm{add}(b) = \{r\}, \mathrm{del}(b) = \{p\}$
  - $\mathrm{pre}(c) = \{q, r\}, \mathrm{add}(c) = \{g\}, \mathrm{del}(c) = \{\}$
- $\mathrm{add}(a_{-1}) = \{p, q\}$
- $\mathrm{pre}(a_n) = \{g\}$

A plan *π* for *D* is a sequence of actions that when applied to the initial state, lead to a state wherein the goal is satisfied.

Definition 2. *A plan $\boldsymbol{\pi} = (\boldsymbol{a_{-1}}, \boldsymbol{a_0}, \dots, \boldsymbol{a_{n-1}}, \boldsymbol{a_n})$ in a complete domain D is a sequence of actions that corresponds to a sequence of states $(\boldsymbol{a_0}, \dots, \boldsymbol{s_n})$, where*

- $s_0 = add(a_{-1})$
- $pre(a_t) \subseteq s_t\ for\ t = 0, \dots, n$
- $s_{t+1} = s_t \backslash del(a_t) \cup add(a_t)\ for\ t = 0, \dots, n-1$

We omit a_{-1} and a_n from the plans in our discussion when appropriate, with the understanding that each plan must use the initial and goal actions.

For example, the plan (*a*, *b*, *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.

Incomplete Planning Domains

Incomplete planning domains are identical to complete planning domains, with the exception that the actions are incompletely specified. Much like planning with incomplete state information [14], 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 precondition 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, and discuss this choice in more detail in the following section.

In the following, we discuss incomplete domains and extend the complete domain model with features for possible preconditions and effects. We note that an incomplete domain corresponds to a set of complete domains, each differing in terms of the inclusion of the possible features.

Definition 3. *An incomplete planning domain* $\widetilde{\boldsymbol{D}}$ *defines the tuple* $(\boldsymbol{P}, \widetilde{\boldsymbol{A}}, \widetilde{\boldsymbol{a}}_{-1}, \widetilde{\boldsymbol{a}}_{n})$, *where:*

- *P is a set of propositions*
- *$\tilde{A}$ is a set of incomplete action descriptions, where each $\tilde{a} \in \tilde{A}$ defines*

  - $pre(\tilde{a}) \subseteq P$, *a set of known preconditions*
  - $\widetilde{pre}(\tilde{a}) \subseteq P$, *a set of possible preconditions*
  - $add(\tilde{a}) \subseteq P$, *a set of known add effects*
  - $\widetilde{add}(\tilde{a}) \subseteq$, *a set of possible add effects*
  - $del(\tilde{a}) \subseteq$, *a set of known delete effects*
  - $\widetilde{del}(\tilde{a}) \subseteq$, *a set of possible delete effects*
- $\tilde{a}_{-1} \subseteq P$ *defines a set of initially true propositions*
- $\tilde{a}_n \subseteq P$ *defines the goal propositions*

Consider the following example of an incomplete domain:

- $P = \{p, q, r, g\}$
- $\tilde{A} = \{\tilde{a}, \tilde{b}, \tilde{c}\}$

$$\begin{array}{lll} \text{pre}(\tilde{a}) = \{p, q\}, & \text{add}(\tilde{a}) = \{\}, & \text{del}(\tilde{a}) = \{\}, \\ \widetilde{\text{pre}}(\tilde{a}) = \{r\}, & \widetilde{\text{add}}(\tilde{a}) = \{r\}, & \widetilde{\text{del}}(\tilde{a}) = \{p\} \end{array}$$

$$\begin{array}{lll} \text{pre}(\tilde{b}) = \{p\}, & \text{add}(\tilde{b}) = \{r\}, & \text{del}(\tilde{b}) = \{p\}, \\ \widetilde{\text{pre}}(\tilde{b}) = \{\}, & \widetilde{\text{add}}(\tilde{b}) = \{\}, & \widetilde{\text{del}}(\tilde{b}) = \{q\} \end{array}$$

$$\begin{array}{lll} \text{pre}(\tilde{c}) = \{r\}, & \text{add}(\tilde{c}) = \{g\}, & \text{del}(\tilde{c}) = \{\}, \\ \widetilde{\text{pre}}(\tilde{c}) = \{q\}, & \widetilde{\text{add}}(\tilde{c}) = \{\}, & \widetilde{\text{del}}(\tilde{c}) = \{\} \end{array}$$

- $\text{add}(\tilde{a}_1) = \{p, q\}$
- $\text{pre}(\tilde{a}_n) = \{g\}$

A plan $\tilde{\pi}$ for $\widetilde{D}$ is a sequence of actions that when applied, *can lead* to a state wherein the goal is satisfied (i.e., the final action’s preconditions are satisfied). This is opposed to a plan $\pi$for *D*, which *does lead* to a state wherein the goal is satisfied.

Definition 4. *A plan* $\boldsymbol{\tilde{\pi}} = (\boldsymbol{\tilde{a}_{-1}}, \boldsymbol{\tilde{a}_0}, \ldots, \boldsymbol{\tilde{a}_{n-1}} \boldsymbol{\tilde{a}_n})$ *in an incomplete domain* $\boldsymbol{\widetilde{D}}$ *is a sequence of actions, that corresponds to a sequence of states* $(\boldsymbol{s_0}, \ldots, \boldsymbol{s_n})$*, where:*

- $s_0 = add(\tilde{a}_{-1})$
- $pre(\tilde{a}_t) \subseteq \mathrm{s_t}$ for $t = 0, \ldots, n$
- $s_{t+1} = s_t \backslash del(\tilde{a}_t) \cup add(\tilde{a}_t) \cup \widetilde{add}(\tilde{a}_t)$ *for* $t = 0, \ldots, n-1$

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.

Definition 5. *The set of incomplete domain features* □$(\boldsymbol{\widetilde{D}})$ *is comprised of the following propositions:*

- $\widetilde{pre}(\tilde{a}, p)$ if $p \in \widetilde{pre}(\tilde{a})$ and $\tilde{a} \in \tilde{A}$
- $\widetilde{add}(\tilde{a}, p)$ if $p \in \widetilde{add}(\tilde{a})$ and $\tilde{a} \in \tilde{A}$
- $\widetilde{del}(\tilde{a}, p)$ if $p \in \widetilde{del}(\tilde{a})$ and $\tilde{a} \in \tilde{A}$

Each incomplete domain feature *f* ∈ □ can result in a different type of plan fault (aligning with GL's original naming conventions):

- Open precondition fault $\mathsf{OP}(\tilde{a}, p)$: if $\widetilde{\mathrm{pre}}(\tilde{a}, p) \in$ □$(\widetilde{D})$ and $\tilde{a}$ is applied to a state *s* where *p* is not true.
- Unlisted effect fault $\mathsf{UE}(\tilde{a}, p)$: if $\widetilde{\mathrm{add}}(\tilde{a}, p) \in$ □$(\widetilde{D})$ and after $\tilde{a}$ is applied, *p* is a precondition for another action.
- Possible clobberer fault $\mathsf{PC}(\tilde{a}, p)$: if $\widetilde{\mathrm{del}}(\tilde{a}, p) \in$ □$(\widetilde{D})$ and after $\tilde{a}$ is applied, *p* is not reestablished by another action and *p* is precondition.

In this sense, each type of incomplete domain features can cause a plan fault if said type can directly or indirectly prevent achievement of a subsequent action's precondition. Each subset of □ corresponds to an interpretation of the incomplete domain.

Definition 6. *An interpretation D^i of the incomplete domain* $\tilde{\boldsymbol{D}}$ *is defined with respect to a subset of the incomplete domain features* $\boldsymbol{F^i} \subseteq$ □ *so that:*

- $P^i = P$
- $a_n^i = a_n$
- $a_{-1}^i = a_{-1}$
- *For each* $\tilde{a} \in \tilde{A}$ *there exists an* $a \in A^i$ *where*
    - $pre(a) = pre(\tilde{a}) \cup \{p | pre(\tilde{a}, p) \in F^i\}$
    - $add(a) = add(\tilde{a}) \cup \{p | add(\tilde{a}, p) \in F^i\}$
    - $del(a) = del(\tilde{a}) \cup \{p | del(\tilde{a}, p) \in F^i\}$

We also refer to the set of incomplete features □$(\tilde{a})$ that are specific to an action $\tilde{a}$ so that □$(\tilde{a}) = \left\{\mathrm{pre}(\tilde{a}, p) \middle| \mathrm{pre}(\tilde{a}, p) \in \ (\tilde{D})\right\} \cup \left\{\mathrm{add}(\tilde{a}, p) \middle| \mathrm{add}(\tilde{a}, p) \in \ (\tilde{D})\right\}$ $\cup \left\{\mathrm{del}(\tilde{a}, p) | \mathrm{del}(\tilde{a}, p) \in (\tilde{D})\right\}$.

For example, the complete domain example from the previous section is an interpretation of the incomplete domain above, where $F^0 = \{\mathrm{add}(\tilde{a}, r), \mathrm{pre}(\tilde{c}, q)\}$.

Definition 4 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. In this sense, we ensure that the plan is valid for the least constraining (most optimistic) interpretation of the incomplete domain. As we show, we can determine the interpretations in which a plan is invalid and use the number of such failed interpretations as a plan quality metric.

COMPARISON OF POSSIBLE ACTION FEATURES WITH LOCAL CLOSED WORLDS

Definition 3 defines incomplete actions by sets of respective known and possible preconditions and effects. GL define incomplete actions similar to STRIPS actions (Definition 1) with additional local closed world statements of the form `DoesNotRelyOn`$(\tilde{a}, p)$ (p is not a precondition of $\tilde{a}$) or `CompletePreconditions`$(\tilde{a})$ (the preconditions of $\tilde{a}$ are known).

We note that these representations are equivalent if we consider the set of known, possible, and impossible preconditions (and similarly for effects) of actions. For example, `CompletePreconditions`$(\tilde{a})$ is equivalent to stating $\widetilde{pre}(\tilde{a}) = \{\}$ (i.e., the set of possible preconditions is empty). Likewise, `DoesNotRelyOn`$(\tilde{a}, p)$ is equivalent to stating $p \notin \widetilde{pre}(\tilde{a})$, and that for all $q \in P$, the lack of a statement `DoesNotRelyOn`$(\tilde{a}, q)$ is equivalent to stating $q \in \widetilde{pre}(\tilde{a})$ (i.e., impossible preconditions are not possible preconditions, and not impossible preconditions are possible preconditions).

While the representations are equivalent, the obvious question is whether one is more succinct than the other. The answer largely depends on the problem being modeled. See Table 1 for examples. Notice that the sizes of the representations are equivalent when stating, for example, that an action has complete preconditions; we either record the fact that the preconditions are complete or that the set of possible preconditions is empty. The difference is with respect to stating, for example, that an individual proposition is not a

Table 1: Examples of Comparing Representations.

	Definition 3	GL
$\tilde{a}$ has only the known preconditions p, q.	$\mathrm{pre}(\tilde{a}) = \{p, q\}$, $\widetilde{\mathrm{pre}}(\tilde{a}) = \{\}$	$\mathrm{pre}(\tilde{a}) = \{p, q\}$, $\mathtt{CompletePreconditions}(\tilde{a})$
$\tilde{a}$ has possible precondition r, but z is neither a known nor a possible precondition	$\mathrm{pre}(\tilde{a}) = \{\}$, $\widetilde{\mathrm{pre}}(\tilde{a}) = \{r\}$	$pre(\tilde{a}) = \{\}$, $\mathtt{DoesNotRelyOn}(\tilde{a}, z)$

precondition of an action. Under our representation (Definition 3), the set of possible preconditions would not contain a proposition, and under the GL representation it must be stated that the proposition is not a precondition. However, if a proposition is a possible precondition to an action, we would record it as a possible precondition, and GL would record nothing. As such, the issue comes down to whether there are many possible or impossible preconditions and effects. Our representation is smaller with many impossible features, and GL is smaller with many possible features.

While we describe actions in the grounded (propositional) form, another practical concern is that we use PDDL [15] action schemas to encode problems. Under the GL representation, extending PDDL action schemas to state impossible preconditions (or effects) could require additional action schema parameters that refer to constants in predicates that are not preconditions. If there are many impossible preconditions, the action schemas could mention many additional parameters, which would lead to difficulty when grounding the schemas. We intuit that possible action features are likely to share parameters with known action features and extending PDDL to support our representation would lead to fewer additional action schema parameters. Furthermore, if there are many impossible features, our representation does not mention these features and therefore does not need to reference their parameters in the PDDL action schemas.

DIAGNOSING FAULTS IN PLANS FOR INCOMPLETE DOMAINS

An incomplete plan $\tilde{\pi}$ must achieve the goals associated with optimistic semantics (i.e., possible preconditions need not be satisfied, possible delete effects can be ignored, and possible add effects will occur), but we would prefer that plans succeed under more pessimistic conditions. To quantify the extent to which our optimism is misleading, we introduce and expand upon GL's definitions of risks, which we refer to as faults. A *fault* is a threat to the plan's causal proof that is introduced because of our optimism/ignorance of the underlying domain description. For example, by assuming that possible delete effects do not occur, we introduce a fault when the possible delete effect does in fact delete a required subgoal. By assuming the optimistic semantics, we allow plans that we would not otherwise consider, but by computing the faults, we quantify the level to which the plan is susceptible to failure. The challenge to computing faults is that incomplete action features may have a delayed impact on the plan or no impact at all, and we must determine if they are faults (i.e., guarantee plan failure if the incompleteness manifests unfavorably).

Instead of reviewing GL's definitions, we take a new approach to develop the definitions of faults. We intuit that plans with faults are best analyzed within the framework of *model-based diagnosis* [6, 7], in other words, abductive reasoning using a model of the system. Among all of the techniques developed within model-based diagnosis [6], the most beneficial is a clear characterization of multiple-faults. In contrast, GL discusses only single-faults, which they call risks, and which do not explain plan failures that may occur because of multiple, interacting incomplete domain features. For example, GL would consider a subgoal that is established by two different actions, each of which is subject to disjoint faults, as having no faults. However, by using multiple-faults to explain failure to achieve the proposition, we see that the faults (at least one for each action) interact. Clearly, single-faults are important for identifying a single-point-of-failure, but ignoring multiple-faults could lead to an overly optimistic assessment of a plan. In the following, we generalize GL's notions of faults from singletons to sets, which we call diagnoses.

Model-Based Diagnosis

In defining the diagnoses of plan failure, we draw upon many well established techniques in model-based diagnosis (MBD) [6, 7]. Viewing the plan as a physical system, faults are sets of potentially faulty components that describe anomalous behavior, such as an action not having its preconditions satisfied or a goal not being achieved.

There are two terms from MBD that enable us to describe which sets of faults may cause plan failure. The first term, a *conflict set* [6], is a set of faults in which if at least one of the faults occurs, it can explain the anomalous behavior. A conflict set is inherently disjunctive because any non-empty subset of the conflict set can explain the failure, and it is not required that all components are faulty. The second term, a *diagnosis*, is a set of system components in which every component must be faulty to explain the behavior. In contrast with a conflict set, a diagnosis is conjunctive – every component in the diagnosis must be faulty. However, there may be multiple diagnoses, and each diagnosis is a hypothesis explaining failure. Because of their respective disjunctive and conjunctive semantics, conflict sets and diagnoses can be expressed by the prime implicants (conjunction of propositions that cannot by subsumed by another conjunction of propositions) of a propositional sentence capturing knowledge of the faulty system.

The author of [7] (henceforth abbreviated, Reiter) formulates MBD within a system that is defined by a system description SD and system components COMP, taking the respective forms of first-order sentences and a finite set of constants. The system description includes a distinct unary predicate $\mathrm{AB}(\cdot)$ that indicates abnormal behavior on the part of a system component. For example, the sentence $\mathrm{ANDG}(x) \wedge \neg\mathrm{AB}(x) \rightarrow \mathtt{out}(x)$ = and $(\mathtt{in1}(x),\mathtt{in2}(x))$ indicates that an and-gate that is not abnormal will have its output equal to the logical and of its two inputs. Along with the system description, OBS is an observation of the system's behavior. For example, OBS may contain the facts $\mathtt{out}(and_1) = 0$, $\mathtt{in1}(and_1) = 1$, $\mathtt{in2}(and_1) = 1$, which is anomalous.

Reiter defines approaches to finding conflict sets and diagnoses that rely on refutation proofs. Showing that $\mathrm{SD} \cup \mathrm{OBS} \cup \{\neg\mathrm{AB}(c_1), ..., \neg\mathrm{AB}(c_n)\}$ is inconsistent means that $c_1, ..., c_n$ functioning normally does not explain OBS. That is, $\{c_1, ..., c_n\}$ is a conflict set, a subset of which is to blame for the observation, and at least one of the conflict set components is faulty. For example, $\mathrm{SD} \cup \mathrm{OBS} \cup\{\neg\mathrm{AB}(and_1)\}$ is inconsistent, and $\{and_1\}$ is a conflict set. Reiter also shows that we can refine the conflict sets to include only those components that are mentioned in the refutation proof tree, so that if $\mathrm{SD} \cup \mathrm{OBS} \cup\{\neg\mathrm{AB}(c_1), ...,\neg\mathrm{AB}(c_n)\}$ is inconsistent, but only if $\{\neg\mathrm{AB}(c_i), ...,\neg\mathrm{AB}(c_j)\} \subseteq \{\neg\mathrm{AB}(c_1), ...,\neg\mathrm{AB}(c_n)\}$ appear in the refutation proof, then $\{\neg\mathrm{AB}(c_i), ...,\neg\mathrm{AB}(c_j)\}$ is a conflict set that subsumes $\{\neg\mathrm{AB}(c_1), ...,\neg\mathrm{AB}(c_n)\}$.

A generate-and-test approach is a possible, but naive, method to finding all conflict sets, as it is too inefficient for systems with large numbers of components. Additionally, upon finding all conflict sets one can compute all diagnoses. Reiter defines a diagnosis as a minimal hitting set on the collection of minimal conflict sets; a hitting set x on a collection of sets C is a set wherein for each set $c \in C$, $c \cap x \neq \{\}$. A minimal hitting set x is a set wherein no proper subset $x' \subset x$ is a hitting set. In our small example, $\{and_1\}$ is the only conflict set, making $\{and_1\}$ the only diagnosis. In a more complex scenario wherein the minimal conflict sets are $\{c_1, c_2\}$ and $\{c_1, c_3\}$, the diagnoses are $\{c_1\}$ and $\{c_2, c_3\}$.

Diagnosing Plan Faults in Incomplete Domains

We describe a plan with a set of clauses $\mathtt{SD}(\tilde{\pi})$ and introduce a hypothetical observation that the goal action cannot be executed, $\mathtt{OBS} = \neg \tilde{a}_n$, to determine if a set of incomplete domain features is a conflict set.

Recall that a conflict set is a set of components, of which some subset must be behaving abnormally to explain an anomalous observation. In diagnosing plan faults, a conflict set is comprised of incomplete domain features. However, there exists an asymmetry among the types of incomplete domain features because the absence of a possible add effect in the true domain can cause failure, but the presence of a possible precondition or possible delete effect can cause plan failure. As such, conflict sets (and diagnoses) refer to negative literals for possible add effects and positive literals for possible preconditions and delete effects.

In diagnosing plan faults, conflict sets and diagnoses are of the form $\left\{\neg\widetilde{\text{add}}(\tilde{a},p),\ldots,\neg\widetilde{\text{add}}\left(\tilde{a}',p'\right),\widetilde{\text{pre}}\left(\tilde{b},q\right),\ldots,\widetilde{\text{pre}}\left(\tilde{b}',q'\right),\widetilde{\text{del}}(\tilde{c},r),\ldots,\widetilde{\text{del}}\left(\tilde{c}',r'\right)\right\}$, indicating the absence of possible add effects or the presence of possible preconditions or delete effects causes plan faults. Thus, following the approach of Reiter, if $\tilde{a}_{-1} \cup SD(\tilde{\pi}) \cup \neg a_n$ $\cup \left\{\widetilde{\text{add}}(\tilde{a},p),\ldots,\widetilde{\text{add}}\left(\tilde{a}',p'\right),\neg\widetilde{\text{pre}}\left(\tilde{b},q\right),\ldots,\neg\widetilde{\text{pre}}\left(\tilde{b}',q'\right),\neg\widetilde{\text{del}}(\tilde{c},r),\ldots,\neg\widetilde{\text{del}}\left(\tilde{c}',r'\right)\right\}$ is inconsistent, then $\left\{\neg\widetilde{\text{add}}(\tilde{a},p),\ldots,\neg\widetilde{\text{add}}\left(\tilde{a}',p'\right),\widetilde{\text{pre}}\left(\tilde{b},q\right),\ldots,\widetilde{\text{pre}}\left(\tilde{b}',q'\right),\widetilde{\text{del}}(\tilde{c},r),\ldots\widetilde{\text{del}}\left(\tilde{c}',r'\right)\right\}$ or a subset of it is a conflict set.

We find it more convenient to formulate an equivalent inference task $\tilde{a}_{-1} \cup SD(\tilde{\pi}) \cup \left\{\widetilde{\text{add}}(\tilde{a},p),\ldots,\widetilde{\text{add}}\left(\tilde{a}',p'\right),\neg\widetilde{\text{pre}}\left(\tilde{b},q\right),\ldots,\neg\widetilde{\text{pre}}\left(\tilde{b}',q'\right),\neg\widetilde{\text{del}}(\tilde{c},r),\ldots,\neg\widetilde{\text{del}}(\tilde{c}',r')\right\} \vDash a_n$, and use a theorem prover that is based on modus ponens and negation as failure. In the following section, we make use of the intuitions developed in this section using modus ponens (we show that negation as failure can be made unnecessary) to motivate a forward-chaining state-space planner.

The system description $SD(\tilde{\pi})$ consists of clauses that define the semantics of plans in incomplete domains, which includes conditions under which an action will have its preconditions satisfied and its effects will change the current state. This subsection i) presents the system description and maps it to the original definitions of plans for incomplete planning problems, ii) shows how the system description can be simplified without loss of generality, and iii) describes how an *assumption-based truth maintenance system* (ATMS) [6] can support more efficient diagnosis computation.

Plan System Description

The system description $\mathsf{SD}(\tilde{a})$ is listed in Table 2. The clauses include conditions under which actions are successfully executed, and conditions under which a proposition will be true as a result of applying an action. The clauses can be understood as stating: i) actions require their preconditions to be satisfied *but also require the previous action to be successful*, ii) add effects are proven if the action is proven, iii) possible add effects are proven if the action is executed and the possible add effect is actually an add effect, iv) propositions that are possibly deleted will in fact be true if they were previously true and either the action fails or they are in fact not deleted, and v) all non-deleted propositions are true if they were previously true.

Table 2: The Plan System Description SD($\tilde{\boldsymbol{\pi}}$),.

i)	$\tilde{a}_{t+1}$	$\leftarrow$	$\tilde{a}_t \wedge \left(\bigwedge_{p \in \mathrm{pre}(\tilde{a}_{t+1})} p_{t+1} \right) \wedge \left(\bigwedge_{p \in \widetilde{\mathrm{pre}}(\tilde{a}_{t+1})} (p_{t+1} \vee \neg \widetilde{\mathrm{pre}}(\tilde{a}_{t+1}, p) \right)$	$t = -1 \ldots n-1$
ii)	p_{t+1}	$\leftarrow$	$\tilde{a}_t$	for all $p \in \mathrm{add}(\tilde{a}_t)$
iii)	p_{t+1}	$\leftarrow$	$\tilde{a}_t \wedge \widetilde{\mathrm{add}}(\tilde{a}_t, p)$	for all $p \in \widetilde{\mathrm{add}}(\tilde{a}_t)$
iv)	p_{t+1}	$\leftarrow$	$p_t \wedge \left(\neg \tilde{a}_t \vee \neg \widetilde{\mathrm{del}}(\tilde{a}_t, p) \right)$	for all $p \in \widetilde{\mathrm{del}}(\tilde{a}_t)$
v)	p_{t+1}	$\leftarrow$	p_t	for all $p \in P \backslash \left(\mathrm{del}(\tilde{a}_t) \cup \widetilde{\mathrm{del}}(\tilde{a}_t) \right)$

The system description of the example plan $(\tilde{a}, \tilde{b}, \tilde{c})$ from example 1 is as follows:

$\tilde{a}_{-1} \rightarrow p_0$	$\tilde{a}_0 \wedge p_1 \rightarrow \tilde{b}_1$	$\tilde{b}_1 \wedge q_2 \wedge r_2 \rightarrow \tilde{c}_2$
$\tilde{a}_{-1} \rightarrow q_0$	$\neg\tilde{b}_1 \wedge q_1 \rightarrow q_2$	$\tilde{b}_1 \wedge \neg\widetilde{pre}(\tilde{c},q) \wedge r_2 \rightarrow \tilde{c}_2$
$\tilde{a}_{-1} \wedge p_0 \wedge q_0 \wedge r_0 \rightarrow \tilde{a}_0$	$q_1 \wedge \neg\widetilde{del}(\tilde{b},q) \rightarrow q_2$	$\tilde{c}_2 \rightarrow g_3$
$\tilde{a}_{-1} \wedge p_0 \wedge q_0 \wedge \neg\widetilde{pre}(\tilde{a},r) \rightarrow \tilde{a}_0$	$\tilde{b}_1 \rightarrow r_2$	$q_2 \rightarrow q_3$
$\neg\tilde{a}_0 \wedge p_0 \rightarrow p_1$	$r_1 \rightarrow r_2$	$r_2 \rightarrow r_3$
$p_0 \wedge \neg\widetilde{del}(\tilde{a},p) \rightarrow p_1$		$\tilde{c}_2 \wedge g_3 \rightarrow \tilde{a}_3$
$\tilde{a}_0 \wedge \widetilde{add}(\tilde{a},r) \rightarrow r_1$		
$q_0 \rightarrow q_1$		

We note that the only non-definite clauses correspond to the cases wherein an action fails to execute and thus cannot possibly clobber the corresponding possibly deleted proposition (e.g., $\tilde{a}$ possibly deletes *p*, and we include the clause$\neg\tilde{a}_0 \wedge p_0 \rightarrow p_1$). As we show below, we can simplify the system description to remove such clauses. For all other clauses, we can create definite clauses by replacing each negated literal $\neg f_i$ by a positive literal nf_i.

We establish the correctness of the system description with the following theorem that states that a plan is valid in an interpretation D^i of an incomplete domain if and only if $a_{-1} \cup SD(\tilde{\pi}) \cup F^i$ entails a_n, where $F^i = \{f | f \in F^i\} \cup \{\neg f | f \notin F^i\}$.

Theorem 7: $a_{-1} \cup SD(\tilde{\boldsymbol{\pi}}) \cup F^i \vDash a_{\boldsymbol{n}}$ *iff* $\tilde{\boldsymbol{\pi}}$ *is a plan interpretation of* D^i.

Simplifying the system description for the domain interpretation where $F^0 = \{\widetilde{add}(\tilde{a},r), \widetilde{pre}(\tilde{c},q)\}$, we obtain $SD^0(\tilde{\pi})$:

$\tilde{a}_{-1} \rightarrow p_0$	$\tilde{a}_0 \wedge p_1 \rightarrow \tilde{b}_1$	$\tilde{b}_1 \wedge q_2 \wedge r_2 \rightarrow \tilde{c}_2$

$\tilde{a}_{-1} \rightarrow q_0$	$\neg\tilde{b}_1 \land q_1 \rightarrow q_2$	$c_2 \rightarrow g_3$
$\tilde{a}_{-1} \land p_0 \land q_0 \land r_0 \rightarrow \tilde{a}_0$	$q_1 \rightarrow q_2$	$q_2 \rightarrow q_3$
$\tilde{a}_{-1} \land p_0 \land q_0 \rightarrow \tilde{a}_0$	$\tilde{b}_1 \rightarrow r_2$	$r_2 \rightarrow r_3$
$\neg\tilde{a}_0 \land p_0 \rightarrow p_1$	$r_1 \rightarrow r_2$	$\tilde{c}_2 \land g_3 \rightarrow \tilde{a}_3$
$p_0 \rightarrow p_1$		
$\tilde{a}_0 \rightarrow r_1$		
$q_0 \rightarrow q_1$		

Upon inspection, it is possible to see that $\tilde{a}_{-1} \cup \mathrm{SD}^{i}(\tilde{\pi}) \vDash \tilde{a}_3$.

If we examine all subsets of the incomplete features □, for example $\{\widetilde{pre}(\tilde{a},r), \widetilde{del}(\tilde{a},p), \widetilde{add}(\tilde{a},r), \widetilde{del}(\tilde{b},q), \widetilde{pre}(\tilde{c},q)\}$ where $\tilde{a}_{-1} \cup \mathrm{SD}(\tilde{\pi}) \cup \{\neg\widetilde{\mathrm{pre}}(\tilde{a},r), \neg\widetilde{\mathrm{del}}(\tilde{a},p), \widetilde{\mathrm{add}}(\tilde{a},r), \neg\widetilde{\mathrm{del}}(\tilde{b},q), \neg\widetilde{\mathrm{pre}}(\tilde{c},q)\} \vDash \tilde{a}_3$, we can determine the minimal conflict sets. In our example, we can derive the following minimal conflict sets:

$\{\widetilde{pre}(\tilde{a},r), \widetilde{del}(\tilde{a},p), \widetilde{del}(\tilde{b},q)\}$

$\{\widetilde{pre}(\tilde{a},r), \widetilde{del}(\tilde{a},p), \widetilde{pre}(\tilde{c},q)\}$

From the conflict sets, we determine the following diagnoses:

$\{\widetilde{pre}(\tilde{a},r)\}$

$\{\widetilde{del}(\tilde{a},p)\}$

$\{\widetilde{del}(\tilde{b},q), \widetilde{pre}(\tilde{c},q)\}$

The diagnoses are cases that will guarantee plan failure, if the first action $\tilde{a}$ can fail because of an open precondition fault. The second action $\tilde{b}$ can fail because its precondition p is deleted by $\tilde{a}$ due to a possible clobberer fault. The third action $\tilde{c}$ can fail if its possible precondition q is required (an open precondition fault) and the second action $\tilde{b}$ possibly deletes q (a possible clobberer fault).

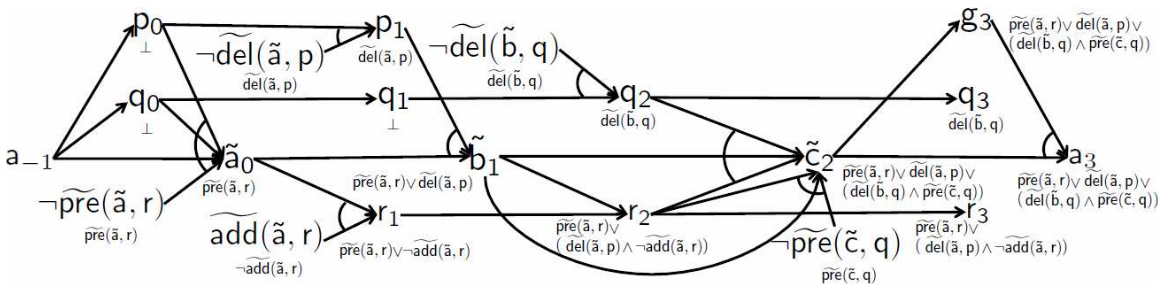

Figure 1: Fault labels and proof trees for the system description of plan $(\tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}})$.

Figure 1 depicts several proof trees for the query

$\mathrm{a}_{-1} \cup \mathrm{SD}'(\tilde{\pi}) \cup \{\neg\widetilde{\mathrm{pre}}(\tilde{\mathrm{a}},\mathrm{r}), \neg\widetilde{\mathrm{del}}(\tilde{\mathrm{a}},\mathrm{p}), \widetilde{\mathrm{add}}(\tilde{\mathrm{a}},\mathrm{r}), \neg\widetilde{\mathrm{del}}(\tilde{\mathrm{b}},\mathrm{q}), \neg\widetilde{\mathrm{pre}}(\tilde{\mathrm{c}},\mathrm{q})\} \vDash \mathrm{a}_3$. In Figure 1, the nodes represent the literals used in the query, and the directed hyper-edges denote clauses. Edges connected by a curved arc denote a conjunction of the antecedents. The propositional sentence annotations can be safely ignored until we discuss the use of the ATMS below. Figure 1 shows that multiple proofs – r_2 and $\tilde{\mathrm{c}}_2$are both proven by two clauses, making a total of four distinct proofs. Each proof relies on a different set of faults not being present; therefore, if any subset of the faults materializes, the proof will fail – these sets of faults correspond to the conflict sets:

$$\{\widetilde{pre}(\tilde{a}, r), \widetilde{del}(\tilde{a}, p), \widetilde{del}(\tilde{b}, q)\}$$

$$\{\widetilde{pre}(\tilde{a}, r), \widetilde{del}(\tilde{a}, p), \widetilde{pre}(\tilde{c}, q)\}$$

$$\{\widetilde{pre}(\tilde{a}, r), \widetilde{del}(\tilde{a}, p), \neg\widetilde{add}(\tilde{a}, r), \widetilde{del}(\tilde{b}, q)\}$$

$\{\widetilde{pre}(\tilde{a},r), \widetilde{del}(\tilde{a},p), \neg\widetilde{add}(\tilde{a},r), \widetilde{pre}(\tilde{c},q)\}$

However, the last two conflict sets are not minimal because they are subsumed by one of the other conflict sets. The minimal conflict sets are:

$\{\widetilde{pre}(\tilde{a},r), \widetilde{del}(\tilde{a},p), \widetilde{del}(\tilde{b},q)\}$

$\{\widetilde{pre}(\tilde{a},r), \widetilde{del}(\tilde{a},p), \widetilde{pre}(\tilde{c},q)\}$

which allows us to compute the following diagnoses (minimal hitting sets):

$\{\widetilde{pre}(\tilde{a},r)\}$

$\{\widetilde{del}(\tilde{a},p)\}$

$\{\widetilde{del}(\tilde{b},q), \widetilde{pre}(\tilde{c},q)\}$

Truth Maintenance Systems

The generate-and-test method of computing conflict sets involves selecting all possible sets of literals F denoting incomplete features and determining if $a_{-1} \cup SD'(\tilde{\pi}) \cup F \vDash a_n$. An alternative is to employ an *assumption-based truth maintenance system* (ATMS) [16], which is a way to represent beliefs (assumptions) and their dependencies. We do this so that we can simultaneously compute all possible proofs for all possible sets F. The approach is to record a label for each literal that is proven to denote a set of contexts relevant to that literal. In our scenario, the contexts denote sets of incomplete domain features F that will prevent the proof of a literal. In the following, we present the definitions of the labels independent of any particular representation, but we describe the implementation of operations required for two alternative representations (prime implicants or OBDDs) in the empirical evaluation.

To represent and compute the contexts preventing the proof of each literal, we recall that each diagnosis is a conjunction of literals $\left\{\neg\widetilde{\mathrm{add}}(\tilde{\mathrm{a}},\mathrm{p}),\ldots,\neg\widetilde{\mathrm{add}}\left(\tilde{\mathrm{a}}',\mathrm{p}'\right),\widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}},\mathrm{q}\right),\ldots,\widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}}',\mathrm{q}'\right),\widetilde{\mathrm{del}}(\tilde{\mathrm{c}},\mathrm{r}),\ldots,\widetilde{\mathrm{del}}\left(\tilde{\mathrm{c}}',\mathrm{r}'\right)\right\}$ where every conjunction must be true in order to cause failure. As such, a label denoting diagnoses can be represented as a disjunction of diagnoses. In the ATMS, we must label each possible premise with the diagnoses preventing its derivation. The possible premises include the initial action a_{-1}, and elements from the set $\left\{\widetilde{\mathrm{add}}(\tilde{\mathrm{a}},\mathrm{p}),\ldots,\widetilde{\mathrm{add}}\left(\tilde{\mathrm{a}}',\mathrm{p}'\right),\neg\widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}},\mathrm{q}\right),\ldots,\neg\widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}}',\mathrm{q}'\right),\neg\widetilde{\mathrm{del}}(\tilde{\mathrm{c}},\mathrm{r}),\ldots,\neg\widetilde{\mathrm{del}}\left(\tilde{\mathrm{c}}',\mathrm{r}'\right)\right\}$, and the labels are defined as

$l(\mathrm{a}_{-1}) = \bot$

$l\left(\widetilde{\mathrm{add}}(\tilde{\mathrm{a}},\mathrm{p})\right) = \neg\widetilde{\mathrm{add}}(\tilde{\mathrm{a}},\mathrm{p}) \ldots l\left(\widetilde{\mathrm{add}}\left(\tilde{\mathrm{a}}',\mathrm{p}'\right)\right) = \neg\widetilde{\mathrm{add}}\left(\tilde{\mathrm{a}}',\mathrm{p}'\right)$

$l\left(\neg\widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}},\mathrm{q}\right)\right) = \widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}},\mathrm{q}\right) \ldots l\left(\neg\widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}}',\mathrm{q}'\right)\right) = \widetilde{\mathrm{pre}}\left(\tilde{\mathrm{b}}',\mathrm{q}'\right)$

$l\left(\neg\widetilde{\mathrm{del}}(\tilde{\mathrm{c}},\mathrm{r})\right) = \widetilde{\mathrm{del}}(\tilde{\mathrm{c}},\mathrm{r}) \ldots l\left(\neg\widetilde{\mathrm{del}}\left(\tilde{\mathrm{c}}',\mathrm{r}'\right)\right) = \widetilde{\mathrm{del}}(\tilde{\mathrm{c}}',\mathrm{r}')$.

The label of the initial action is $\bot$ (logical false) to denote that there is no diagnosis under which the initial action cannot be derived. The label of each literal denoting an incomplete domain feature is the negation of the literal to denote that the only diagnosis under which the literal is not proven is when the literal is not true initially.

All other literals are proven by one or more clauses, and we associate with each clause $h\colon \mathrm{q}_1,\ldots,\mathrm{q}_m \rightarrow \mathrm{p}$ that proves p a sentence $l(h,\mathrm{p}) = l\left(\mathrm{q}_1\right) \vee \ldots \vee l\left(\mathrm{q}_m\right)$ to denote that the clause will fail to prove p in any case where at least *one of* (hence the disjunction) its antecedents is not proven. Multiple clauses h_1, ..., h_k may prove p, allowing us to define $l(\mathrm{p}) = l(h_1,\mathrm{p}) \wedge \ldots \wedge l(h_k,\mathrm{p})$ denoting that p will be unproven if *all of* (hence the conjunction) the clauses fail to prove p.

Figure 1 depicts the labels associated with each literal by the propositional sentence underneath the literal. Consider the incomplete system label $l(\tilde{\mathrm{c}}_2)$, that is proven by two clauses, $h_3 \colon \tilde{\mathrm{b}}_1, \mathrm{q}_2, \mathrm{r}_2, \rightarrow \tilde{\mathrm{c}}_2$ and $h_4 \colon \tilde{\mathrm{b}}_1, \neg \mathrm{f}_4, \mathrm{r}_2 \rightarrow \tilde{\mathrm{c}}_2$. The labels for each of the antecedents of the clauses are as follows:

$$l(\tilde{\mathrm{b}}_1) = \mathrm{f}_0 \vee \mathrm{f}_1$$

$$l(\mathrm{q}_2) = \mathrm{f}_3$$

$$l(\mathrm{r}_2) = \mathrm{f}_0 \vee (\mathrm{f}_1 \wedge \mathrm{f}_2)$$

$$l(\neg \mathrm{f}_4) = \mathrm{f}_4$$

Allowing us to compute for each clause

$$l(h_3, \tilde{\mathrm{c}}_2) = l(\tilde{\mathrm{b}}_1) \vee l(\mathrm{q}_2) \vee l(\mathrm{r}_2) = (\mathrm{f}_0 \vee \mathrm{f}_1) \vee (\mathrm{f}_3) \vee \left(\mathrm{f}_0 \vee (\mathrm{f}_1 \wedge \mathrm{f}_2)\right) = \mathrm{f}_0 \vee \mathrm{f}_1 \vee \mathrm{f}_3$$

$$l(h_4, \tilde{\mathrm{c}}_2) = l(\tilde{\mathrm{b}}_1) \vee l(\neg \mathrm{f}_4) \vee l(\mathrm{r}_2) = (\mathrm{f}_0 \vee \mathrm{f}_1) \vee (\mathrm{f}_4) \vee \left(\mathrm{f}_0 \vee (\mathrm{f}_1 \wedge \mathrm{f}_2)\right) = \mathrm{f}_0 \vee \mathrm{f}_1 \vee \mathrm{f}_4$$

and define

$$l(\tilde{\mathrm{c}}_2) = l(h_3, \tilde{\mathrm{c}}_2) \wedge l(h_4, \tilde{\mathrm{c}}_2) = (\mathrm{f}_0 \vee \mathrm{f}_1 \vee \mathrm{f}_3) \wedge (\mathrm{f}_0 \vee \mathrm{f}_1 \vee \mathrm{f}_4) = \mathrm{f}_0 \vee \mathrm{f}_1 \vee (\mathrm{f}_3 \wedge \mathrm{f}_4)$$

By counting the models of the label $l(\mathrm{g}_3)$, of the goal, it is possible to determine how many interpretations of the incomplete domain will fail to achieve the goal with the plan. In this example, there are 32 interpretations, and 26 will fail to achieve the goal.

Counting Models and Diagnoses

The labels computed in the previous section identify those combinations of incomplete domain features that can prohibit a plan from satisfying the goals, i.e., the models of the labels are interpretations of the incomplete domain that will fail. From the labels, it is possible to compute exactly how many interpretations of the incomplete domain features lead to a successful or unsuccessful plan. Thus, counting the number of domains that will not successfully achieve the goals can be reduced to counting the models of the goal action label $l(\tilde{a}_n)$ (a propositional sentence). The planner described in the next section is based on the idea of using an ATMS to represent plans, and many of its subroutines involve comparing propositional sentences. In comparing a propositional sentence □ with another, we refer to its set of models *M*(□), its set of prime implicants *PI*(□), and its set of *k*-element prime implicants PI_k(□).

While counting models requires polynomial time when a propositional sentence is represented by an OBDD, it requires exponential time when represented by prime implicants. However, we note that the number of prime implicants can be indicative of the number of models, and simply counting the number of prime implicants can provide a heuristic measure.

Referencing the example in the previous section, the three prime implicants $f_0 \vee f_1 \vee (f_3 \wedge f_4)$ have 26 models, whereas the two prime implicants $f_0 \vee f_1$ have 24 models: the number of prime implicants, in this case, is proportional to the number of models. While the relationship between prime implicants and models does not hold in general, we can use the number of prime implicants to heuristically compare propositional sentences (estimating the number of models).

Another observation is that having fewer prime implicants of smaller cardinality can result in fewer models. For example, both $f_0 \vee f_1$ and $f_0 \vee (f_3 \wedge f_4)$ have two prime implicants, but the former has 24 models and the latter has 20 models. Thus, when comparing two propositional sentences, we can compare $|PI_1(\phi)|$ and $|PI_1(\psi)$, and if equal, compare $|PI_2(\phi)|$ and $|PI_2(\psi)|$, and so on, until $|PI_k(\phi)| \neq |PI_k(\psi)|$ for some $k > 0$; if k is the minimum cardinality where $|PI_k(\phi)| < |PI_k(\psi)|$, then we prefer □ (assuming □ represents interpretations of incomplete actions where a plan fails). Thus, we define two preference relations on propositional formulas representing plan failure:

- Model-based: $\phi \prec M \; \psi$ if $|M(\phi)| < |M(\psi)|$
- Diagnosis-based: $\phi \prec PI \; \psi$ if $|PI_k(\phi)| < |PI_k(\psi), k > 0$, and $|PI_j(\phi)| = |PI_j(\psi)|$ for all $j < k$.

In the following, we dispense with the subscripted notation for preference relations, assuming that the context dictates whether the propositional sentences are compared by models or diagnoses.

Comparing the prime-implicants is much less expensive than counting and comparing the number of models, but we may be wrong. Nevertheless, we empirically compare counting OBDD models to counting prime implicants (of different cardinalities) within our planner, and demonstrate significant improvements in planning time with little sacrifice in plan quality when counting prime implicants. Throughout our discussion, when we refer to counting models of □, we assume that □ is represented by an OBDD, and when we refer to counting the prime implicants of □, we assume that □ is already represented by prime implicants. In other words, we assume the representation that is most natural for the type of counting in order to ignore any additional cost of normal form conversion.

FORWARD STATE SPACE PLANNING WITH FAULTS

We present a forward state space planner called `DeFAULT` that uses the approaches developed in the previous section to search for plans that have few faults or few interpretations of the incomplete domain features that result in plan failure. Recall that having few faults and few failed interpretations are connected but rely on counting different quantities (prime implicants or models, respectively). We employ the optimistic semantics for incomplete domain features and extend our state description to capture which incomplete domain features can cause failure to achieve each state proposition; the incomplete features are represented by OBDDs or prime implicants, as in the previous section. We note that computing and representing the prime implicants can be costly; we address this by formulating our approach for any arbitrary, but fixed, bound on the prime implicant cardinality. While the cardinality of each prime implicant is bounded, the number of prime implicants per proposition is indirectly bounded, i.e., there is a finite number of sets with cardinality *k* or less. The impact of bounding the prime implicant cardinality is that we may under-approximate the number of interpretations of the incomplete domain in which the plan will fail.

Adapting the ATMS rules for propagating fault labels to the state space requires some explanation. The most striking differences are that we do not have explicit action literals in the state space and we do not specify the plan semantics by clauses, rather we define the propagation in terms of the state and action descriptions. The lack of action literals and clauses that connect action literals to the goals requires that we track the incomplete domain interpretations that cause plan failure because an action is invalidated, i.e., its preconditions are not satisfied. The state space operations capture the same semantics as the ATMS operations in that we use disjunction to combine faults affecting conjunctive requirements, e.g., action preconditions, and use conjunction to combine disjunctive requirements, e.g., causal support for propositions.

Fault Propagation

In the previous section, we describe how to recursively define the failure explanation (label) for a goal literal, i.e., the propositional models of the label reflect which interpretations fail to achieve the goal. In the following, we discuss rules for the forward propagation of failure explanations to compliment our forward state-space planner.

Initially, we use the explanation $d_{-1}(\tilde{a}_{-1}) = \perp$ to denote that there are no failures affecting the initial state. For all states $s_{t+1}, t \geq 0$, we define:

$$d_{t+1}(p) = \begin{cases} d_t(p) \wedge d_t(\tilde{a}_t) & : p \in \mathrm{add}(\tilde{a}_t) \\ d_t(p) \wedge \left(d_t(\tilde{a}_t) \vee \neg\widetilde{\mathrm{add}}(\tilde{a}_t, p)\right) & : p \in \widetilde{\mathrm{add}}(\tilde{a}_t) \\ \top & : p \in \mathrm{del}(\tilde{a}_t) \\ d_t(p) \vee \widetilde{\mathrm{del}}(\tilde{a}_t, p) & : p \in \widetilde{\mathrm{del}}(\tilde{a}_t) \\ d_t(p) & :\text{otherwise} \end{cases}$$

where the interpretations failing to successfully execute $\tilde{a}_t$ are defined:

$$d_t(\tilde{a}_t) = d_{t-1}(\tilde{a}_{t-1}) \vee \bigvee_{p \in pre(\tilde{a})} d_t(p) \vee \bigvee_{p \in \widetilde{pre}(\tilde{a})} \left(d_t(p) \wedge \widetilde{pre}(\tilde{a}_t, p)\right)$$

In the above, note the correspondence to the ATMS propagation rules. The definition of $d_{t+1}(p)$ refers to the combination of the assumptions of two ATMS clauses, one describing the persistence of p and the other describing the action adding p. Delete effects are false (failed to be true) under all interpretations, which corresponds to the lack of clauses that can prove the proposition in the previous section. Propositions given as possible delete effects have their faults defined as any faults previously affecting the proposition or a fault introduced when it is in fact deleted. Propositions not affected by the action persist their faults, corresponding to the persistence clauses in the previous section. Any interpretation in which one of the action's preconditions are unsatisfied will cause the action to fail, and any interpretation in which the most previous action or any prior action fails will cause the plan to fail.

Finally, to count the number of interpretations under which a plan fails, we count the models of $d(\tilde{\pi}) = d_n(\tilde{a}_n)$, which expresses the interpretations wherein any of the actions did not have its preconditions satisfied or the goal was not satisfied. Recall that we require valid plans to achieve the goal under the optimistic semantics, so we are guaranteed that if $\text{pre}(a_n) \subseteq s_n$, the plan will succeed in at least one interpretation of the incomplete domain.

As an aside, it is possible to determine the interpretations that fail to successfully execute the plan up to and including time *t* by computing $d_t(\tilde{a}_t)$. We also note that as long as *n* is the earliest time that the goal is achieved, we are guaranteed that $d_t(\tilde{a}_t) \vDash d(\tilde{\pi})$. That is, because $\tilde{a}_n$ is required in the plan, the definition of $d_t(\tilde{a}_t)$ for $t = 0, \dots, n-1$ may include failures due to relevant or irrelevant (not directly or indirectly causally supporting the goals) prior actions.

We illustrate the fault propagation for the example plan, as follows.

Example 2. Consider the fault propagation required for our example plan $(\tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}})$. Initially, the explanation for the initial action is $\boldsymbol{d_{-1}}(\tilde{\boldsymbol{a}}_{\boldsymbol{-1}}) = \perp$, and state $\boldsymbol{s_0} = \{\boldsymbol{p}, \boldsymbol{q}\}$ is labeled as follows:

$d_0(p) = \perp$

$d_0(q) = \perp$

$d_0(r) = \top$

$d_0(g) = \top$

After applying $\tilde{a}$ to s_0, we attain the state $s_1 = \{p, q, r\}$ with the following explanations:

$$\begin{aligned}
d_0(\tilde{a}) &= d_0(p) \vee d_0(q) \vee \big(d_0(r) \wedge \widetilde{\mathrm{pre}}(\tilde{a}, r)\big) \\
&= \perp \vee \perp \vee \big(\top \wedge \widetilde{\mathrm{pre}}(\tilde{a}, r)\big) \\
&= \widetilde{\mathrm{pre}}(\tilde{a}, r) \\
d_1(p) &= d_0(p) \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \\
&= \perp \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \\
&= \widetilde{\mathrm{del}}(\tilde{a}, p) \\
d_1(q) &= d_0(q) = \perp \\
d_1(r) &= d_0(r) \wedge \Big(d_0(\tilde{a}) \vee \neg \widetilde{add}(\tilde{a}, r)\Big) \\
&= \top \wedge \Big(\widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \neg \widetilde{\mathrm{add}}(\tilde{a}, r)\Big) \\
&= \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \neg \widetilde{\mathrm{add}}(\tilde{a}, r) \\
d_1(g) &= d_0(g) = \top
\end{aligned}$$

Applying $\tilde{b}$ to s_1 results in the state $s_2 = \{q, r\}$, with the explanations:

$$\begin{aligned}
d_1(b) &= d_0(\tilde{a}) \vee d_1(p) = \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \\
d_2(p) &= \top \\
d_2(q) &= d_1(q) \vee \widetilde{\mathrm{del}}(\tilde{b}, q) \\
&= \bot \vee \widetilde{\mathrm{del}}(\tilde{b}, q) \\
&= \widetilde{\mathrm{del}}(\tilde{b}, q) \\
d_2(r) &= d_1(r) \wedge d_1(\tilde{b}) \\
&= \left(\widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \neg\widetilde{\mathrm{add}}(\tilde{a}, r)\right) \wedge \left(\widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p)\right) \\
&= \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \left(\neg\widetilde{\mathrm{add}}(\tilde{a}, r) \wedge \widetilde{\mathrm{del}}(\tilde{a}, p)\right) \\
d_2(g) &= d_1(g) = \top
\end{aligned}$$

Finally, after applying $\tilde{c}$ to s_2, we compute $s_3 = \{q, r, g\}$ and the explanations:

$$\begin{aligned}
d_2(\tilde{c}) &= d_1(\tilde{b}) \vee d_2(r) \vee \left(d_2(q) \wedge \widetilde{\mathrm{pre}}(\tilde{c}, q)\right) \\
&= \left(\widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p)\right) \vee \left(\widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \left(\neg\widetilde{\mathrm{add}}(\tilde{a}, r) \wedge \widetilde{\mathrm{del}}(\tilde{a}, p)\right)\right) \\
&\qquad \vee \left(\widetilde{\mathrm{del}}(\tilde{b}, q) \wedge \widetilde{\mathrm{pre}}(\tilde{c}, q)\right) \\
&= \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \vee \left(\widetilde{\mathrm{del}}(\tilde{b}, q) \wedge \widetilde{\mathrm{pre}}(\tilde{c}, q)\right) \\
d_3(p) &= d_2(p) = \top \\
d_3(q) &= d_2(q) = \widetilde{\mathrm{del}}(\tilde{b}, q) \\
d_3(r) &= d_2(r) = \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \left(\neg\widetilde{\mathrm{add}}(\tilde{a}, r) \wedge \widetilde{\mathrm{del}}(\tilde{a}, p)\right) \\
d_3(g) &= d_2(g) \wedge d_2(\tilde{c})
\end{aligned}$$

$$= \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \vee \left(\widetilde{\mathrm{del}}(\tilde{b}, q) \wedge \widetilde{\mathrm{pre}}(\tilde{c}, q)\right)$$

The plan results in the following failure diagnosis:

$$d(\tilde{\pi}) = d_3(\tilde{a}_3) = d_3(g) \vee d_2(\tilde{c}) = \widetilde{\mathrm{pre}}(\tilde{a}, r) \vee \widetilde{\mathrm{del}}(\tilde{a}, p) \vee \left(\widetilde{\mathrm{del}}(\tilde{b}, q) \wedge \widetilde{\mathrm{pre}}(\tilde{c}, q)\right)$$

Forward State-Space Planning

`DeFAULT` is a forward state-space planner that is based on Downward, and its greedy best first search algorithm. `DeFAULT` compares partial plans only in terms of their heuristic value (described in the next section). While `DeFAULT` does not compare the faults introduced by plan prefixes leading to states on the fringe of the search, these faults are used in the heuristic computation.

PLANNING GRAPH FAULT PROPAGATION

Similar to propagating faults in a plan, we can propagate faults in the relaxed planning problem to compute a heuristic measure of the faults affecting goal achievement. We start with a brief description of heuristics in complete domains.

Planning Graph Heuristics

A relaxed planning graph is a layered graph of sets of vertices $(P_t, A_t, \ldots, A_{t+m}, P_{t+m+1})$. The planning graph built w.r.t. a state s_t defines $P_t = s_t, A_{t+k} = \{a | pre(a) \subseteq P_{t+k}, a \in A \cup A(P)\}$, and $P_{t+k+1} = \{p | a \in A_{t+k}, p \in \mathrm{add}(a)\}$, for $k = 0, \ldots, m$. The set $A(P)$ includes noop actions for each proposition, such that $A(P) = \{a_p | p \in P, \mathrm{pre}(a_p) = \mathrm{add}(a_p) = p, \mathrm{del}(a_p) = \emptyset\}$. A simple heuristic, h^+ for the number of actions to achieve the goal $\mathrm{pre}(a_n)$ from s_t is equivalent to the minimum level *k* where the goal propositions are reached, $h^+ = \min_{\mathrm{k:G \subseteq P_{t+k}}} k$. The h^{FF} heuristic [17] solves this relaxed planning problem by choosing actions from A_{t+m} to support the goals in P_{t+m+1}, and recursively for each chosen action's preconditions, counting the number of chosen actions.

Diagnoses

When planning in incomplete domains, we would like to minimize the number of interpretations of the incomplete domain under which the plan fails. A heuristic should measure and attempt to minimize the number of failed interpretations in the estimated suffix of a plan. As in the state space, we propagate information about failed interpretations in the planning graph to estimate the quality of a plan completion starting in the current state.

Propagating faults in the planning graph resembles propagating faults over the plan. The primary difference is how we reconcile the faults for a proposition when the proposition has multiple sources of support. In a level of the relaxed planning graph, there are potentially many sources of support for a proposition, and we simply select the supporter with the preferred set of faults, either a fewer number of models or preferred set of prime implicants. The chosen supporting action, denoted $\hat{a}_{t+k}(p)$, determines the faults affecting a proposition p at level$t + k + 1$.

A relaxed planning graph with propagated faults is a layered graph of sets of vertices of the form $\left(\hat{P}_t, \hat{A}_t, \ldots, \hat{A}_{t+m}, \hat{P}_{t+m+1}\right)$. The relaxed planning graph built w.r.t. a state $\tilde{s}_t$ defines $\hat{P}_0 = \tilde{s}_t, \hat{A}_{t+k} = \left\{a \middle| \mathrm{pre}(\tilde{a}) \subseteq \hat{P}_{t+k}, \tilde{a} \in \tilde{A} \cup A(P)\right\}$ and $\hat{P}_{t+1} = \left\{p \middle| \tilde{a} \in \hat{A}_t, p \in \mathrm{add}(\tilde{a}) \cup \widetilde{\mathrm{add}}(\tilde{a})\right\}$, for $k = 0, \ldots, m$. 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 explain below, it associates faults with the possible adds. Each proposition p has associated faults, denoted $\hat{d}_t(p)$. Each action also has associated faults, denoted $\hat{d}_{t+k}(\tilde{a})$. The faults $\hat{d}_t(p)$ affecting a proposition are defined by its supporting action $\hat{a}_{t+k}(p)$, such that $\hat{d}_t(p) = d_t(p)$, and for $k = 0,1, \ldots \hat{d}_{t+k+1}(p) =$

$$\left(\bigwedge_{p \in \mathrm{add}\left(\hat{a}_{t+k}(p)\right)} \hat{d}_{t+k}\left(\hat{a}_{t+k}(p)\right)\right) \wedge \left(\bigwedge_{p \in \widetilde{\mathrm{add}}\left(\hat{a}_{t+k}(p)\right)} \hat{d}_{t+k}\left(\hat{a}_{t+k}(p)\right) \vee \neg \widetilde{\mathrm{add}}(\hat{a}_{t+k}(p), p)\right)$$

and the faults affecting an action are defined by the faults for the action's preconditions,

$$\hat{d}_{t+k}(\tilde{a}) = \left(\bigvee_{p \in \mathrm{pre}(\tilde{a})} \hat{d}_{t+k}(p) \right) \vee \left(\bigvee_{p \in \widetilde{\mathrm{pre}}(\tilde{a})} \hat{d}_{t+k}(p) \wedge \widetilde{\mathrm{pre}}(\tilde{a}, p) \right)$$

Propositions in the planning graph initially have the same faults associated with them as in state $\tilde{s}_t$ and are defined by $d_t(\cdot)$. Every action in every level k of the planning graph can be invalidated by any fault affecting its preconditions, or by open precondition faults. Beyond the initial level, faults affecting a proposition include faults that invalidate its supporting actions or are associated with unlisted effects supporting the proposition.

We note that the rules for propagating faults in the planning graph differ from the rules for propagating faults in the state space. In the state space, the action failure explanations include explanations for any prior action failing. In the relaxed planning problem, the action failure explanations include only explanations affecting the action's preconditions, and not prior actions. In the relaxed planning problem, 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 $t + k + 1$ when one of the following conditions is met: i) the planning graph reaches a fix-point where the labels do not change, $\hat{d}_{t+k}(p) = \hat{d}_{t+k+1}(p)$ for all p, or ii) the goals have been reached at $t + k + 1 - c$ (c levels after the goals are first reached) and the fixed point has not yet been reached. The heuristic $h^{\sim M}$ measures the number of interpretations that fail to reach the goals in the last level such that $h^{\sim M} = \left|M\left(\bigvee_{g\in\mathrm{pre}(\tilde{a}_n)} \hat{d}_{t+m+1}(g)\right)\right|$, where $m+1$ is the last level of the planning graph. Similarly, $h^{\sim PI}$stores the set of prime implicants $\bigvee_{g\in\mathrm{pre}(\tilde{a}_n)} \hat{d}_{t+m+1}(g)$, and uses the preference relation for prime implicants to compare search nodes. The $h^{\sim FF}$ heuristic makes use of the chosen supporting actions $\hat{a}_{t+k}(p)$ for each proposition that requires support in the relaxed plan, and, hence, measures the number of actions used while attempting to minimize fault. DeFAULT uses both heuristics, treating $h^{\sim FF}$ as the primary heuristic and using $h^{\sim M}$ or $h^{\sim PI}$ to break ties.

EMPIRICAL EVALUATION

The empirical evaluation is divided into three sections: the domains used for the experiments, the test setup used, and a discussion of the results. We compare `DeFAULT` with a control planner that uses the same search algorithm and implementation, but uses the FF heuristic to guide search. We attempted but do not compare with the PFF [18] CPP planner because of some unresolved stability issues. The questions that we sought to answer include:

- Can a classical planner (that ignores action incompleteness) find reasonable quality solutions in incomplete domains?
- How well does a planner that counts failure explanation models scale?
- Can a planner that counts prime implicants in failure explanations scale well and find high quality solutions?
- Does bounding the size of prime implicants lead to better planner performance without harming plan quality?

Domains

We use five domains in the evaluation: a modified Pathways, Bridges, Blind Navigator, a modified PARC Printer, and BarterWorld. In Pathways, we derived multiple instances by randomly injecting incomplete domain features, with probabilities 0.0, 0.01, 0.25, 0.5, 0.75, and 1.0 for each type of fault and for each action. In the other domains, we injected incomplete domain features with a probability of 0.5. All results are the average of ten random instances of each problem. The Pathways domain from the international planning competition involves actions that model chemical reactions in signal transduction pathways. Pathways is a naturally incomplete domain wherein 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 domains consist of a traversable grid, and the task is to find different treasure at each corner of the grid. There are three versions in which each subsequent version has an additional type of incompleteness. In Bridges1, a bridge might be required to cross between some grid locations and can cause open precondition faults. In Bridges2, many of the bridges may have a troll living underneath that will take all the treasure accumulated, and cause a possible clobberer fault. In Bridges3, some of the corners may give additional treasures, causing unlisted effect faults.

In Blind Navigator we must navigate from one corner of a grid to the opposite corner. Unfortunately, when traveling from one square to the next, there is a possibility of getting lost (a possible clobberer fault). In order to reorient oneself, it is possible to observe two types of landmarks that are either highly or lowly observable. A highly observable landmark supports certain localization, and a lowly observable landmark *may* support localization (an unlisted effect fault).

The PARC Printer domain from the international planning competition 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 (unlisted effects), and traveling between some locations may require certain items (possible preconditions) and may result in the loss of an item (possible delete). The instances involve different size grids and number of items.

Test Setup and `DeFAULT` Implementation

The tests were run on a machine running Linux with a 3 Ghz Xeon processor, a memory limit of 2GB, and a time limit of 20 minutes per run. All code (aside from POND) was written in Java and run on the 1.6 JVM. Both `DeFAULT` and the control planner shared the same greedy best first search implementation that uses deferred heuristic evaluation and a dual-queue for preferred and non-preferred operators [19]. Both planners also used the same planning graph implementation. The planners were 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 in a domain. Those planners that solve more problems can be easily identified, and their overall relative plan quality and efficiency are evident by the cumulative plots.

The `DeFAULT` planner was implemented in Java, and each of the configurations of the planner shared common source code, with the exception of their respective techniques for fault propagation in the state space and heuristic computation.

The first configuration, which we refer to as `DeFAULT`-*FF*, does not compute fault information, making it largely a classical planner that uses the FF heuristic. The one aspect of the `DeFAULT`-*FF* configuration that is not common to classical planners is how it assumes the optimistic semantics for the incomplete domain (ignoring possible preconditions and delete effects, but assuming possible add effects will occur).

The second configuration, based on the prime implicant representation of fault diagnoses, is simply referred to as `DeFAULT`-*k*, where *k* is the bound on the cardinality of the prime implicants. We use values of *k* from one to three. The implementation of the prime implicant fault computations is largely straightforward, i.e., does not employ any non-trivial optimizations. The required conjunction and disjunction operations combine the conjunctive clauses in the standard way, and remove clauses that are subsumed or exceed the cardinality bound.

Based on counting models (domain interpretations), the third configuration is called `DeFAULT`-*All* to highlight the fact that it does not approximate the representation of the faulty domain interpretations. Its representation of the interpretations makes use of the JDD package for OBDDs to implement conjunction, disjunction, and model counting.

Results

We first discuss the results in each domain, and then conclude this section with a discussion of the trends seen across the domains. In several of the domains, we discuss alternative versions of the domain that include increasingly more incompleteness (measured by the number of incomplete features). In all of the results plots, the legend refers to a configuration of the planner *X*, denoting `DeFAULT`-*X* (as described above).

Blind Navigation

Figure 2 shows that the `DeFAULT`-*FF* configuration finds plans of comparable quality to the configurations that reason about incompleteness only in the smallest instances (instances 1-10, which are 2x2 grids).

Each additional ten instances increase the grid size to 4x4, 8x8, and 16x16. The `DeFAULT`-*FF*, and `DeFAULT`-1, -2, or -3 configurations cannot solve instances bigger than 8x8, due to the importance of reasoning about incompleteness in this domain. It

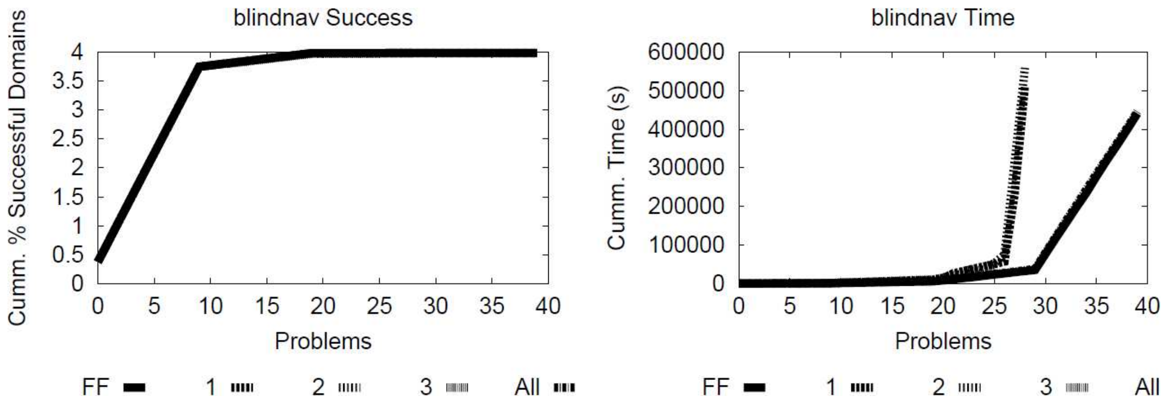

Figure 2: Cumulative quality and time comparison in Blind Navigation domain.

appears that approximating the failed interpretations of the domain does not harm the quality of plans, but it does limit the scalability.

Parc Printer

Figure 3 shows reasoning about incompleteness in the Parc Printer domain is important to finding high quality plans, but not necessarily important to finding plans. The DeFAULT-*FF* configuration scales well, but finds the worst quality plans. The DeFAULT-1, -2, and -3 configurations find the highest quality plans (which are identical quality), but do not scale as well as DeFAULT-*All*. The difference between model counting and prime implicant counting in this domain may be attributed to the potentially efficient OBDD representation of the failed domain interpretations, but fortuitous prime implicant representation that helps identify other, better plans.

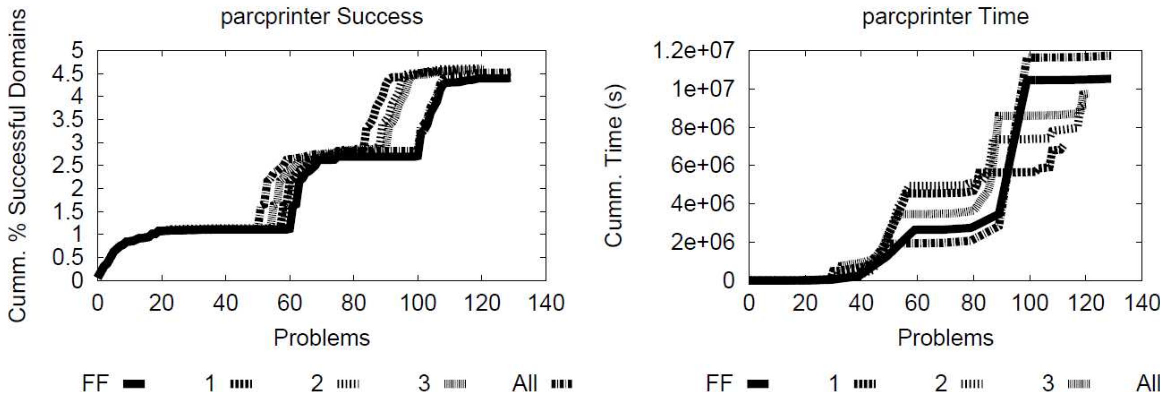

Figure 3: Cumulative quality and time comparison in Parc Printer domain.

Bridges

Figure 4 shows results for all three versions of the domain combined, and Figures 5, 6, and 7 show the results for the respective versions of the domain. Common to all versions of the domain, `DeFAULT`-*FF* finds the poorest quality plans, but surprisingly is not overly superior in terms of planning time and problems solved. In all versions of the domain, the `DeFAULT`-1 configuration solves the most problems, and in the third version of the domain it has the best overall planning time. However, considering more faulty interpretations, either by using `DeFAULT`-1, -2, or *All*, does improve plan quality at the expense of scalability and planning time. Interestingly, the trends remain the same across the versions of the domain, with `DeFAULT`-*FF* performing progressively worse as we include different types of incomplete domain features.

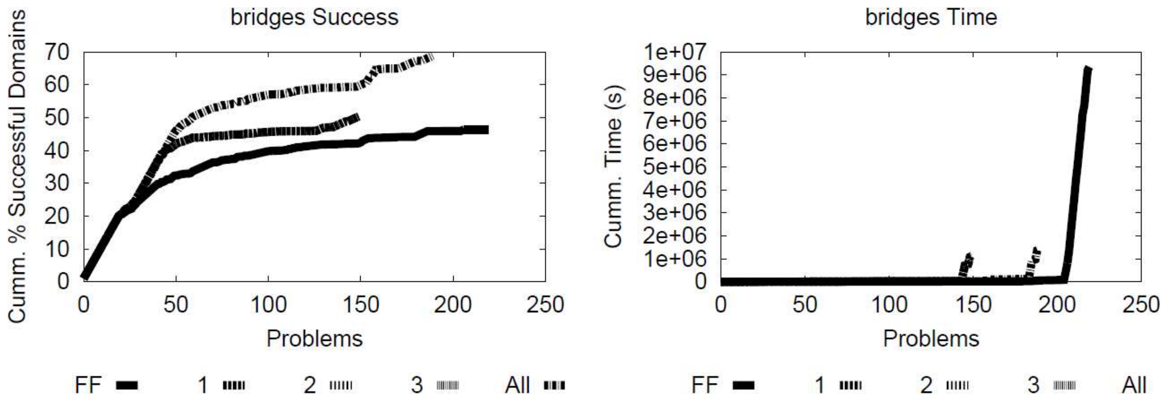

Figure 4: Cumulative quality and time comparison in all three version of the Bridges domain.

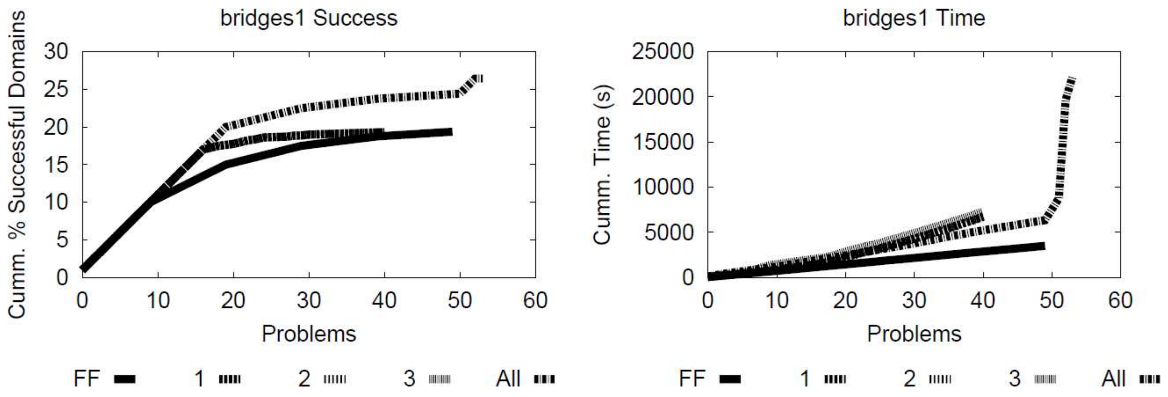

Figure 5: Cumulative quality and time comparison in Bridges1 Domain.

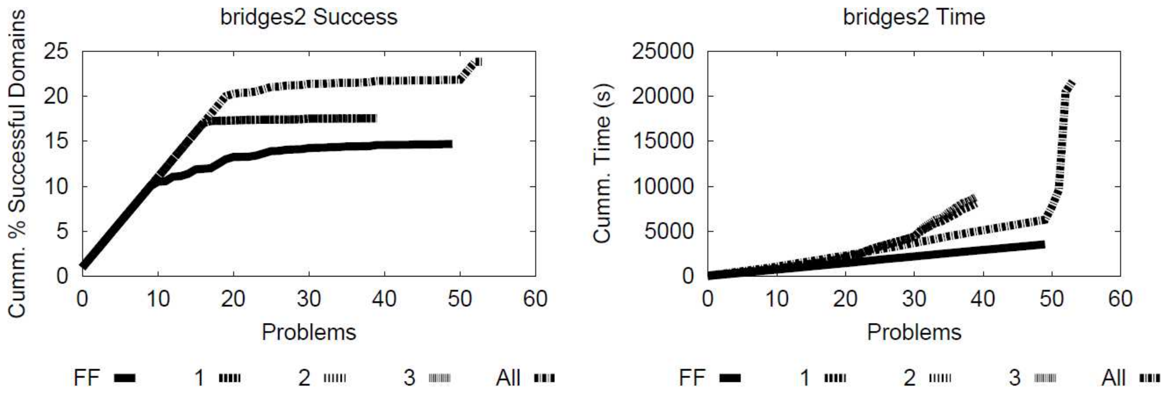

Figure 6: Cumulative quality and time comparison in Bridges2 Domain.

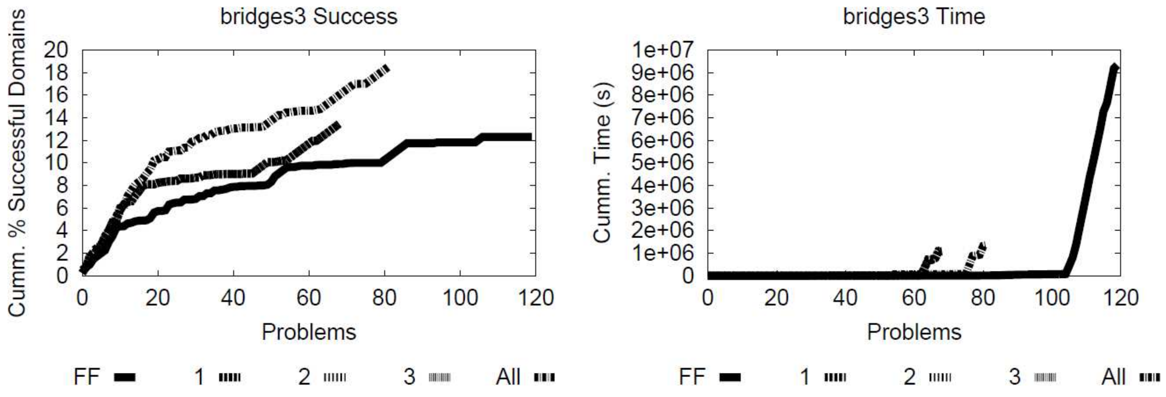

Figure 7: cumulative quality and time comparison in Bridges3 Domain.

Barter World

Figure 8 shows the combined results for four versions of the Barter World domain, which are shown individually in Figures 9, 10, 11, and 12, which respectively set the probability of the domain generator introducing incomplete features to 0.25, 0.5, 0.75, and 1.0.

The trend identified by Figure 8 is that failing to reason about incompleteness permits greater scalability but poor quality plans, and as the reasoning about incompleteness strengthens, so does the plan quality (but at the expense of scalability). As the number of incomplete features grows across Figures 9 to 12, we see the same trend exacerbated: weaker reasoning about incompleteness scales better, and stronger reasoning finds better quality plans.

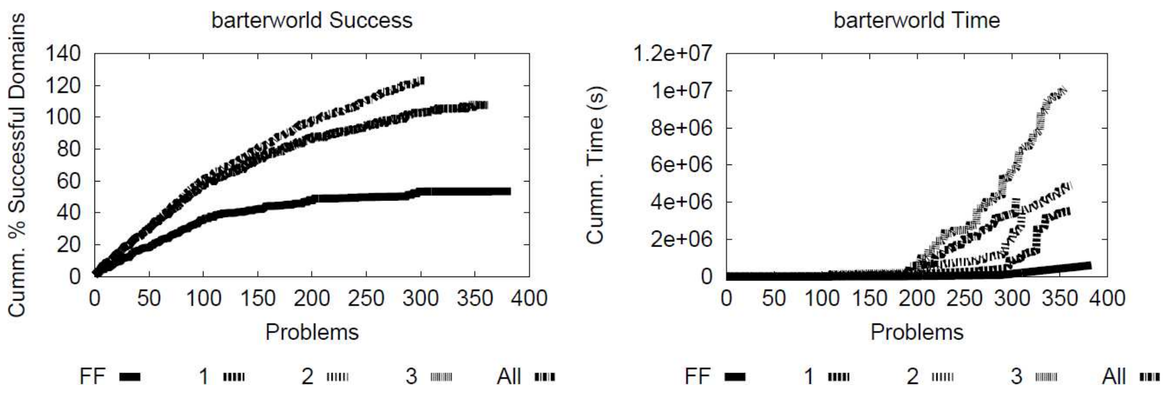

Figure 8: Cumulative quality and time comparison in all instances of Barter World comain.

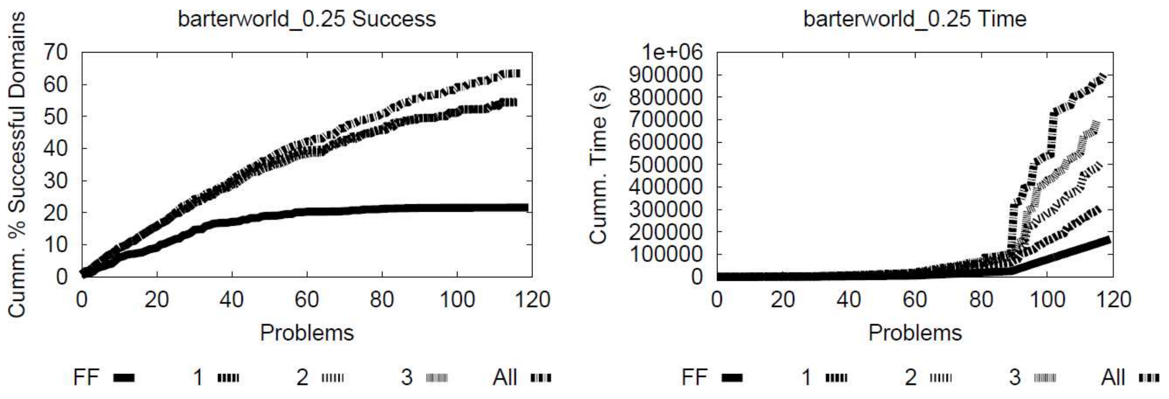

Figure 9: Cumulative quality and time comparison in 0.25 density Barter World comain.

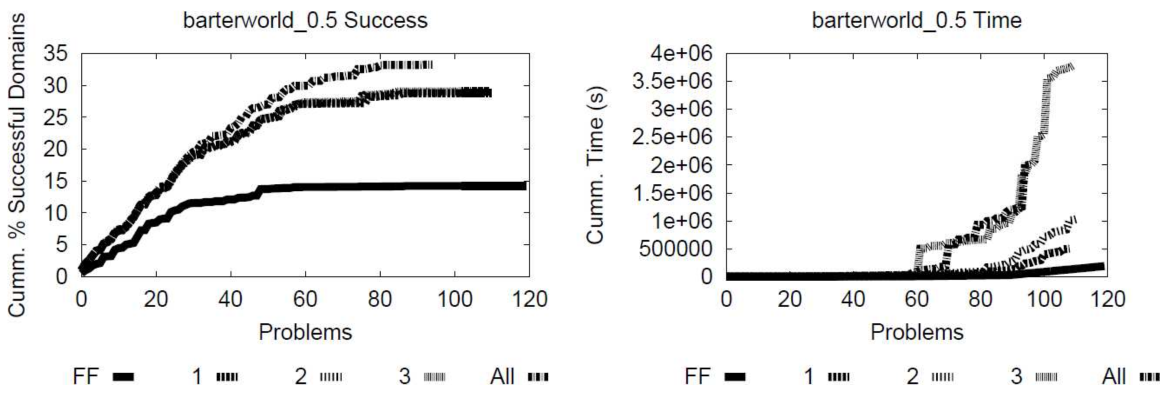

Figure 10: Cumulative quality and time comparison in 0.5 density Barter World domain.

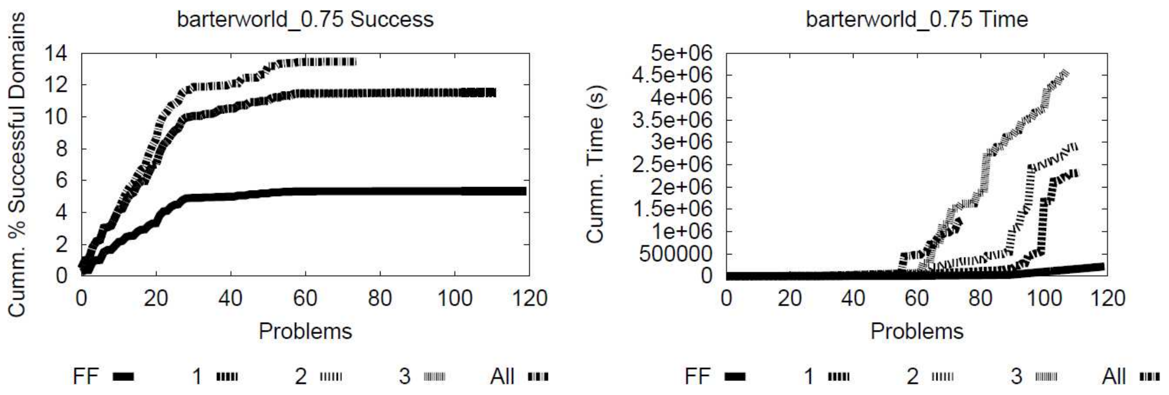

Figure 11: Cumulative quality and time comparison in 0.75 density Barter World domain.

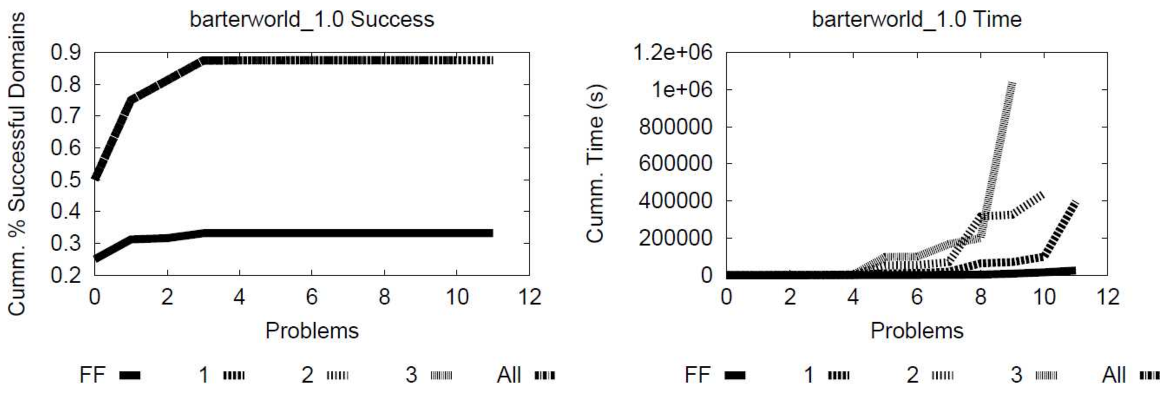

Figure 12: Cumulative quality and time comparison in 1.0 density Barter World Domain.

Pathways

Figure 13 shows the combined results for four versions of the Pathways domain that set the probability of generating incomplete domain features to 0.25, 0.5, 0.75, and 1.0. The results for each of the settings are shown individually in Figures 14, 15, 16, and 17.

The combined results demonstrate that the techniques for reasoning about incompleteness find similar quality plans, but the weaker the technique, the lower its planning time. As the probability of including incomplete features increases, the stronger reasoning about incompleteness does not scale as well, but the quality of the plans found by the techniques is similar.

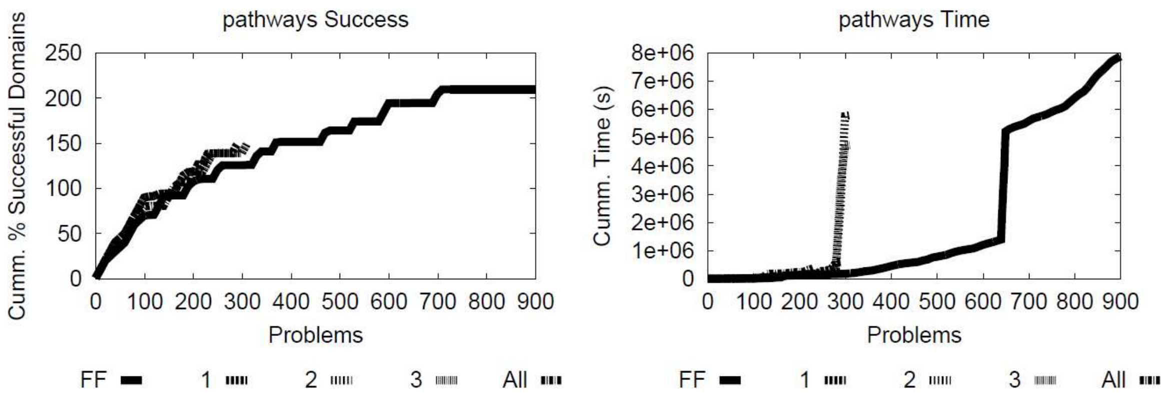

Figure 13: Cumulative quality and time comparison in Pathways domain.

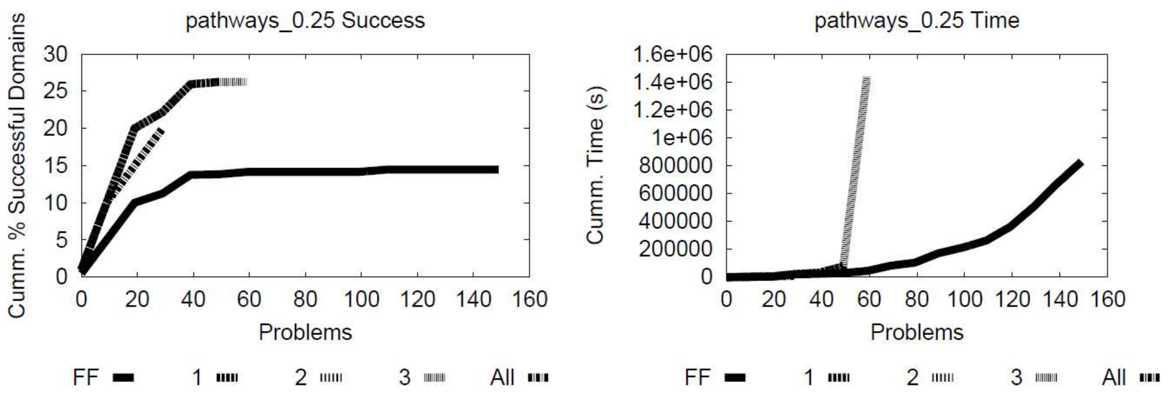

Figure 14: Cumulative quality and time comparison in 0.25 density Pathways domain.

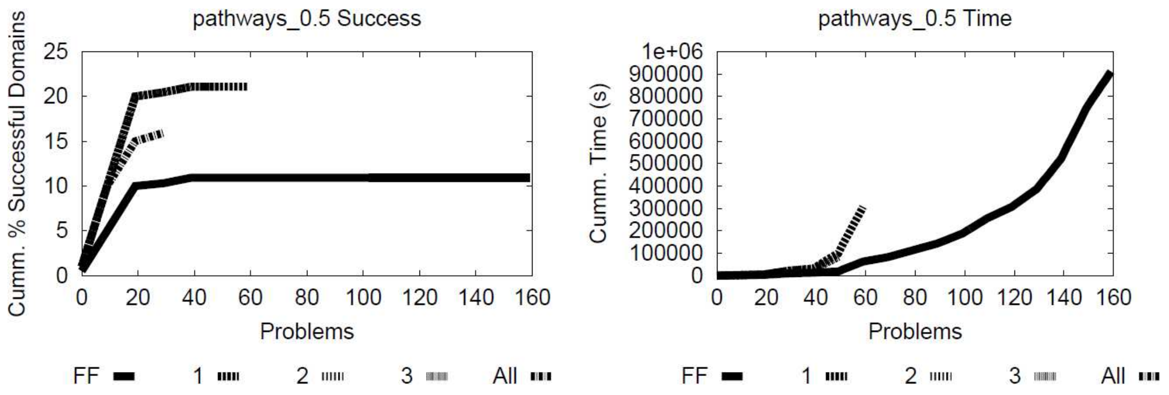

Figure 15: Cumulative quality and time comparison in 0.5 density Pathways domain.

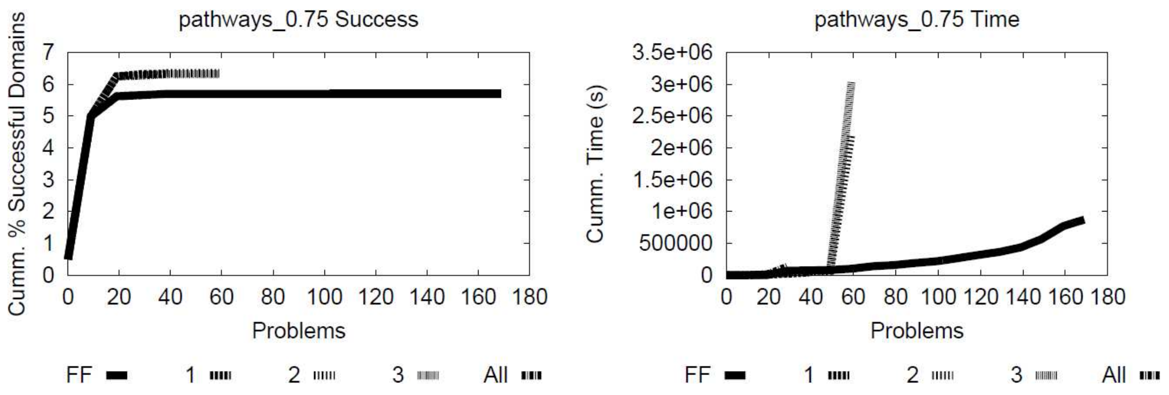

Figure 16: Cumulative cuality and time comparison in 0.75 density Pathways domain.

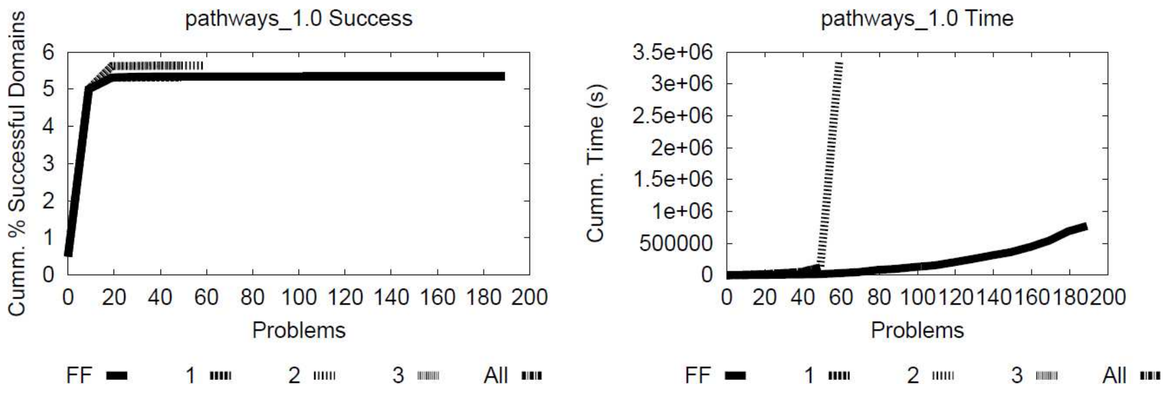

Figure 17: Cumulative quality and time comparison in 1.0 density Pathways domain.

Discussion

As the strength of the reasoning about incompleteness increases from ignoring incompleteness to tracking increasingly higher cardinality prime implicants, to tracking all interpretations of an incomplete domain, we tend to see increasing plan quality, in terms of the number of domain interpretations that will successfully execute the plan and achieve the goal. We also see scalability decrease as a result. Reasoning about prime implicants tends to be a useful middle-ground whereby plans have good quality, and planner scalability is best.

RELATED WORK

Planning with faults is noticeably similar to planning with incomplete information [12], wherein action descriptions instead of states are incomplete. As we have shown, incomplete domains can be translated to CPP domains, and planners such as POND and PFF [18] are applicable. However, while the translation is theoretically feasible, practical issues regarding numeric precision prohibit effective use of existing planners.
Our investigation is an instantiation of model-lite planning [1]. Constraint-based hierarchical task networks are an alternative, pointed out by [1], 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 [5] model for evaluating plans in incomplete domains. Our methods for computing faults are slightly different in that we compute faults in the forward direction and are more specific about which faults occur. In addition to calculating faults of partial plans, we have also presented a relaxed planning heuristic informed by fault.

Prior work of [20] also addresses planning with incomplete models, but focuses on online planning and execution to learn the model, similar to model-based reinforcement learning. We differ in that we assume no feedback from the environment and attempt to find the best plan possible offline. However, the plans found by `DeFAULT` have the potential to guide either knowledge engineers or experimentation.

CONCLUSION

We have presented the first work to address planning in incomplete domains as a heuristic search to find mostly-correct plans. Our planner, `DeFAULT`, i) performs forward search while maintaining sets of plan faults, and ii) estimates the future faults incurred by propagating faults 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 and find much better quality plans. We have also shown that representing explanations of plan failure with prime implicants leads to better scalability than a complete representation using OBDDs and counting models.

REFERENCES

[1] Kambhampati, S. Model-lite planning for the web age masses. In *Proceedings of 22nd National Conference on AI*, 2007.

[2] Wu, K., Yang, Q., and Jiang, Y. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *Knowledge Engineering Review 22*, 2 (2007), 135–152.

[3] Piergiorgio Bertoli, P., Adi Botea, A., and Simone Fratini, Report of the board of judges. In *Third International Competition on Knowledge Engineering for Planning and Scheduling*, 2009.

[4] Mailler, R., Bryce, D., Shen, J., and Orielly, C. Mable: A framework for natural instruction. In *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2009.

[5] Garland, A. and Lesh, N. Plan evaluation with incomplete action descriptions. In *Proceedings of 18th National Conference on Artificial Intelligence*, 2002.

[6] de Kleer, J. and Williams, B.C. Diagnosing multiple faults. *Artificial Intelligence* 32, 1 (1987), 97– 130.

[7] Reiter, R. A theory of diagnosis from first principles. *Artificial Intelligence* 32, 1 (1987), 57–95.

[8] Robertson, J. and Bryce, D. Reachability heuristics for planning in incomplete domains. In *Proceedings of the ICAPS Workshop on Heuristics for Domain Independent Planning*, 2009.

[9] Gerevini, A., Bonet, B., and Givan, R. Report of the committee. In *Fifth International Planning Competition*, 2006.

[10] DRoth, D. On the hardness of approximate reasoning. *Artificial Intelligence 82* 1-2 (Apr. 1996), 273–302.

[11] Darwiche A. and Marquis, P. A knowledge compilation map. *Journal of Artificial Intelligence Research 17* (2002), 229–264.

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

[13] Fikes, R. and Nilsson, N.J. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of Assoc. for the Advancement of Artificial Intelligence,* 1971, 608–620.

[14] Bryce, D., Kambhampati, S., and Smith, S.E. Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research 26* (2006), 35–99.

[15] McDermott, D. Pddl-the planning domain definition language. Technical report, CVC TR-98-003, Yale University Computer Science Department, 1998.Available at: www.cs.yale.edu/homes/dvm.

[16] de Kleer, J. An assumption-based tms. *Artificial Intelligence 28*, 2 (1986), 127–162, 1986.

[17] Hoffmann, J. and Nebel, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research 14* (2001), 253–302.

[18] Domshlak, C. and Hoffmann, J. Fast probabilistic planning through weighted model counting. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2006.

[19] Malte Helmert, M. The fast downward planning system. *Journal of Artificial Intelligence Research 26* (2006),191–246.

[20] Chang, A. and Amir, E. Goal achievement in partially known, partially observable domains. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2006.