Technical Report CS-TR-2797, UMIACS-TR-91-154, SRC-TR-91-96.

Complexity, Decidability and Undecidability Results for Domain-Independent Planning: A Detailed Analysis*

Kutluhan Erol† Dana S. Nau‡ V.S. Subrahmanian§

University of Maryland
College Park, Maryland 20742, U.S.A.

Abstract

In this paper, we examine how the complexity of domain-independent planning with STRIPS-style operators depends on the nature of the planning operators.

We show conditions under which planning is decidable and undecidable. Our results on this topic solve an open problem posed by Chapman [8], and clear up some difficulties with his undecidability theorems.

For those cases where planning is decidable, we show how the time complexity varies depending on a wide variety of conditions:

- whether or not function symbols are allowed;
- whether or not delete lists are allowed;
- whether or not negative preconditions are allowed;
- whether or not the predicates are restricted to be propositional (i.e., 0-ary);
- whether the planning operators are given as part of the input to the planning problem, or instead are fixed in advance.
- whether or not the operators can have conditional effects.

Furthermore, we provide insights about the reasons for our results.

*This work was supported in part by the Army Research Office under Grant Number DAAL-03-92-G-0225, as well as by NSF Grant NSFD CDR-88003012 to the University of Maryland Institute for Systems Research, and NSF grants IRI-8907890 and IRI-9109755.

†Department of Computer Science. Email: kutluhan@cs.umd.edu.

‡Department of Computer Science, Institute for Systems Research, and Institute for Advanced Computer Studies. Email: nau@cs.umd.edu.

§Department of Computer Science and Institute for Advanced Computer Studies. Email: vs@cs.umd.edu.

Contents

Table 1: Decidability of domain-independent planning.$^{\alpha}$

Allow function symbols?	Allow infinitely many constants?	Allow infinite initial states?	Allow delete lists and/or negated preconditions?	Telling whether a plan exists
yes	yes/no	yes/no	yes/no/no$^{\beta}$	semidecidable
	no	no	no$^{\gamma}$	decidable
no	yes	yes	yes/no	semidecidable
		no	yes	semidecidable
			no	decidable
	no	no$^{\delta}$	yes/no	decidable

$^{\alpha}$These results are independent of whether the operators are fixed in advance and/or have conditional effects.
$^{\beta}$No operator has more than one precondition.
$^{\gamma}$With acyclicity and boundedness restrictions as described in Section 3.3.
$^{\delta}$The other restrictions ensure that the initial state will always be finite.

Table 2: Complexity of domain-independent planning.$^{\alpha}$

Language restrictions	How are the operators given?	Allow delete lists?	Allow negated preconditions?	Telling whether a plan exists	Telling whether there is a plan of length $\leq k$
no function symbols, and finitely many constant symbols	given in the input	yes	yes/no	EXPSPACE-comp.	NEXPTIME-comp.
		no	yes	NEXPTIME-comp.	NEXPTIME-comp.
			no	EXPTIME-comp.	NEXPTIME-comp.
			no$^{\beta}$	PSPACE-complete	PSPACE-complete
	fixed in advance	yes	yes/no	PSPACE $^{\delta}$	PSPACE $^{\delta}$
		no	yes	NP $^{\delta}$	NP $^{\delta}$
			no	P	NP $^{\delta}$
			no$^{\beta}$	NLOGSPACE	NP
all predicates are 0-ary (propositions)	given in the input	yes	yes/no	PSPACE-complete$^{\epsilon}$	PSPACE-complete
		no	yes	NP-complete$^{\epsilon}$	NP-complete
			no	P $^{\epsilon}$	NP-complete
			no$^{\beta}$/no$^{\gamma}$	NLOGSPACE-comp.	NP-complete
	fixed in advance	yes/no	yes/no	constant time	constant time

$^{\alpha}$These results are independent of whether the operators have conditional effects.
$^{\beta}$No operator has more than one precondition.
$^{\gamma}$Every operator with more than one precondition is the composition of other operators.
$^{\delta}$With PSPACE- or NP-completeness for some sets of operators.
$^{\epsilon}$Results due to Bylander [5].

1 Introduction

Much planning research has been motivated, in one way or another, by the difficulty of producing complete and correct plans. For example, techniques such as abstraction [29, 7, 28, 38] and task reduction [33, 7, 38] were developed in an effort to make planning more efficient, and concepts such as deleted-condition interactions were developed to describe situations which make planning difficult.

Despite the acknowledged difficulty of planning, it is only recently that researchers have begun to examine the computational complexity of planning problems and the reasons for that complexity [8, 5, 19, 20, 26, 27]. This research has yielded some surprising results. For example, Gupta and Nau [19, 20] have shown that contrary to prior expectations, deleted-condition interactions are easy to handle in blocks-world planning.

Pednault [30] suggests that since planning is intractable in general, researchers should try to identify constraints that will lead to efficient planning. The current paper addresses this goal, by examining how the complexity of domain-independent planning depends on the nature of the planning operators.

We consider planning problems in which the current state is a set of ground atoms, and each planning operator is a STRIPS-style operator consisting of three lists of atoms: a precondition list, an add list, and a delete list. Our results can be summarized as follows:

1. The decidability results are shown in Table 1. If function symbols are allowed, then determining, in general, whether a plan exists[1] is *undecidable* (more specifically, semidecidable).[2] This is true even if we have no delete lists and the precondition list of each operator contains at most one (non-negated) atom. If no function symbols are allowed and only finitely many constant symbols are allowed, then plan existence is *decidable*, regardless of the presence or absence of delete lists and/or negated preconditions.

   Even when function symbols are present, plan existence is decidable if the planning domains being considered have no deletion lists, no negated atoms occur in the precondition list, and the domains satisfy certain acyclicity and bounded-ness properties.

   Whether the planning operators are fixed in advance or given as part of the input, and whether or not they are allowed to have conditional effects, does not affect these results.

2. The complexity results are shown in Table 2. When there are no function symbols and only finitely many constant symbols (so that planning is decidable), the computational complexity varies from constant time to EXPSPACE-complete, depending on the following conditions:

   - whether or not we allow delete lists and/or negative preconditions,
   - whether or not we restrict the predicates to be propositional (i.e., 0-ary),
   - whether we fix the planning operators in advance, or give them as part of the input.

   The presence or absence of conditional operators does not affect these results.

3. We have solved an open problem stated by Chapman in [8]: whether or not planning is undecidable when the language contains infinitely many constants but the initial state is finite. In particular, this problem is decidable in the case where the planning operators have no negative preconditions and no delete lists. If the planning operators are allowed to have negative preconditions and/or delete lists, then the problem is undecidable.

[1] The formal definition of this problem appears in Section 2.

[2] We use "decidable" and "undecidable" interchangably with "recursive" and "recursively enumerable," respectively.

4. Chapman's Second Undecidability Theorem states that "planning is undecidable even with a finite initial situation if the action representation is extended to represent actions whose effects are a function of their input situation" [8], i.e., if the language contains function symbols and infinitely many constants. Our results show that even with a number of additional restrictions, planning is still undecidable.

   We also correct a misimpression about this theorem, which has been thought by some researchers [31, 11] to refer to operators that have conditional effects. It does not—and as we mentioned above, our decidability and complexity results are unaffected by whether or not the operators have conditional effects.

The rest of this paper is organized as follows. Section 2 contains the basic definitions. Section 3 contains the decidability and undecidability results, and Section 4 compares and contrasts them with Chapman's results. Section 5 presents the complexity results. Section 6 discusses the related work. Section 7 contains concluding remarks. Section 7.3 discusses future research directions. The proofs of the theorems and lemmas appear in the appendices.

2 Preliminaries

Researchers in planning have long been interested in planning with STRIPS-style operators, and this interest still continues [5, 8, 19, 26, 27]. In the original STRIPS planner [12], the planning operators' precondition lists, add lists, and delete lists were allowed to contain arbitrary well-formed formulas in first-order logic. However, there were a number of problems with this formulation, such as the difficulty of providing a well-defined semantics for it [23]. Thus, in subsequent work, researchers have placed severe restrictions on the nature of the planning operators [28]. Typically, the precondition lists, add lists and delete lists contain only atoms, and the goal is a conjunct of ground or existentially quantified atoms. Our definitions below are in accordance with such commonly accepted formulations.

Definition 2.1 Let $\mathcal{L}$ be any first-order language generated by finitely many constant symbols, predicate symbols, and function symbols. Then a *state* is any finite set of ground atoms in $\mathcal{L}$.[3]

Intuitively, a state tells us which ground atoms are currently true: if a ground atom A is in state S, then A is true in state S, and if $B \notin S$, then B is false in state S. Thus, a state is simply an Herbrand interpretation for the language $\mathcal{L}$, and hence each formula of first-order logic is either satisfied or not satisfied in S according to the usual first-order logic definition of satisfaction.

Definition 2.2 Let $\mathcal{L}$ be an ordinary first-order language. Then a *planning operator* α is a 4-tuple $(\mathrm{Name}(\alpha), \mathrm{Pre}(\alpha), \mathrm{Add}(\alpha), \mathrm{Del}(\alpha))$, where

1. $\mathrm{Name}(\alpha)$ is a syntactic expression of the form $\alpha(X_1, \ldots, X_n)$ where each X_i is a variable symbol of $\mathcal{L}$;

2. $\mathrm{Pre}(\alpha)$ is a finite set of literals, called the *precondition list* of α, whose variables are all from the set $\{X_1, \ldots, X_n\}$;

[3] It is standard practice to assume that first-order languages contain only finitely many constant symbols, and that states contain only finitely many atoms. However, in order to compare some of our results with Chapman's [8] results, in a few places in this paper we will be interested in violating one or both of these assumptions. When we do so, we will say so explicitly.

(a) initial configuration (b) goal configuration

Figure 1: Initial and goal configurations for five blocks a, b, c, d, e.

3. $\text{Add}(\alpha)$ and $\text{Del}(\alpha)$ are both finite sets of atoms (possibly non-ground) whose variables are taken from the set $\{X_1, \ldots, X_n\}$. $\text{Add}(\alpha)$ is called the *add list* of α, and $\text{Del}(\alpha)$ is called the *delete list* of α.

Observe that negated atoms are allowed in the precondition list, but not in the add and delete lists.

When defining a planning operator α, often $\text{Name}(\alpha)$ will be clear from context. In such cases, we will not always specify $\text{Name}(\alpha)$ explicitly.

Definition 2.3 A *first-order planning domain* (or simply a *planning domain*) is a pair $\mathbf{P} = (S_0, \mathcal{O})$, where S_0 is a state called the *initial state*, and $\mathcal{O}$ is a finite set of planning operators. The *language* of $\mathbf{P}$ is the first-order language $\mathcal{L}$ generated by the constant, function, predicate, and variable symbols appearing in $\mathbf{P}$, along with an infinite number of additional variable symbols.

Definition 2.4 A *goal* is a conjunction of atoms which is existentially closed (i.e., the variables, if any, are existentially quantified).

Definition 2.5 A *planning problem* is a triple $\mathbf{P} = (S_0, \mathcal{O}, G)$, where $(S_0, \mathcal{O})$ is a planning domain and G is a goal.

Example 2.1 (Blocks World) Suppose we want to talk about a blocks-world planning domain in which there are five blocks a, b, c, d, e, along with the "stack", "unstack", "pickup", and "putdown" operators used by Nilsson [28]. Suppose the initial configuration is as shown in Fig. 1(a), and the goal is to have b on c on d, as shown in Fig. 1(b). Then we will define the language, operators, planning domain, and planning problem as follows:

1. The language $\mathcal{L}$ will contain five constant symbols a, b, c, d, e, each representing (intuitively) the five blocks. $\mathcal{L}$ will contain no function symbols, and will contain the following predicate symbols: "handempty" will be a propositional symbol (i.e. a 0-ary predicate symbol), "on" will be a binary predicate symbol, and "ontable", "clear", and "holding" will be unary predicate symbols. In addition, we will have a supply of variable symbols, say, $X_1, X_2, \ldots$. Note that operator names, such as "stack", "unstack", etc., are not part of the language $\mathcal{L}$.

2. The "unstack" operator will be the following 4-tuple:

$$\begin{aligned}
\text{Name(unstack)} &= \text{unstack}(X_1, X_2) \\
\text{Pre(unstack)} &= \{\text{on}(X_1, X_2), \text{clear}(X_1), \text{handempty}()\} \\
\text{Del(unstack)} &= \{\text{on}(X_1, X_2), \text{clear}(X_1), \text{handempty}()\} \\
\text{Add(unstack)} &= \{\text{clear}(X_2), \text{holding}(X_1)\}
\end{aligned}$$

The "stack", "pickup", and "putdown" operators are defined analogously.

3. The planning domain will be $(S_0, \mathcal{O})$, where S_0 and $\mathcal{O}$ are as follows:

$$\begin{aligned} S_0 &= \{\text{clear}(a), \text{on}(a,b), \text{on}(b,c), \text{ontable}(c), \text{clear}(d), \\ &\quad\ \text{on}(d,e), \text{ontable}(e), \text{handempty}()\}; \\ \mathcal{O} &= \{\text{stack}, \text{unstack}, \text{pickup}, \text{putdown}\}. \end{aligned}$$

The planning problem will be $(S_0, \mathcal{O}, G)$, where $G = \{\text{on}(b,c), \text{on}(c,d)\}$.

Definition 2.6 Let $\mathbf{P} = (S_0, \mathcal{O})$ be a planning domain, α be an operator in $\mathcal{O}$ whose name is $\alpha(X_1, \ldots, X_n)$, and θ be a substitution that assigns ground terms to each $X_i, 1 \le i \le n$. Suppose that the following conditions hold for states S and S':

$$\begin{aligned} &\{A\theta : A \text{ is an atom in } \text{Pre}(\alpha)\} \subseteq S; \\ &\{B\theta : \neg B \text{ is a negated literal in } \text{Pre}(\alpha)\} \cap S = \emptyset; \\ &S' = (S - (\text{Del}(\alpha)\theta)) \cup (\text{Add}(\alpha)\theta)\,. \end{aligned}$$

Then we say that α is *θ-executable* in state S, *resulting* in state S'. This is denoted symbolically as

$$S \overset{\alpha,\theta}{\Longrightarrow} S'.$$

Definition 2.7 Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain and G is a goal. A *plan that achieves* G is a sequence $S_0, \ldots, S_n$ of states, a sequence $\alpha_1, \ldots, \alpha_n$ of planning operators, and a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that

$$S_0 \overset{\alpha_1,\theta_1}{\Longrightarrow} S_1 \overset{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \overset{\alpha_n,\theta_n}{\Longrightarrow} S_n \tag{1}$$

and G is satisfied by S_n, i.e. there exists a ground instance of G that is true in S_n. The *length* of the above plan is n.

We will often say that (1) above is a plan that achieves G.

Definition 2.8 Let $\mathbf{P} = (S_0, \mathcal{O})$ be a planning domain or $\mathbf{P} = (S_0, \mathcal{O}, G)$ be a planning problem; and let $\mathcal{L}$ be the language of $\mathbf{P}$. Then

1. α is *positive* if $\text{Pre}(\alpha)$ is a finite set of *atoms* (i.e. negations are not present in $\text{Pre}(\alpha)$).

2. α is *deletion-free* if $\text{Del}(\alpha) = \emptyset$.

3. α is *context-free* if $|\text{Pre}(\alpha)| \le 1$, i.e., α has at most one precondition.

4. α is *side-effect-free* if $|\text{Add}(\alpha) \cup \text{Del}(\alpha)| \le 1$, i.e., α has at most one postcondition.

5. $\mathcal{L}$ is *function-free* if it contains no function symbols.

6. $\mathcal{L}$ is *propositional* if every predicate P in $\mathcal{L}$ is propositional (i.e., 0-ary).

If every operator in $\mathcal{O}$ is positive, deletion-free, context-free, and/or side-effect-free, then we say that $\mathcal{O}$ (and thus $\mathbf{P}$) is, too. Likewise, $\mathbf{P}$ is function-free and/or propositional if $\mathcal{L}$ is. Note that if $\mathbf{P}$ is propositional, then no operator will ever use function symbols, hence it will not matter whether $\mathbf{P}$ is function-free or not.

Definition 2.9 PLAN EXISTENCE is the following problem:

> Given a planning problem $\mathbf{P} = (S_0, \mathcal{O}, G)$, does there exist a plan in $\mathbf{P}$ that achieves G?

Definition 2.10 PLAN LENGTH is the following problem:[4]

> Given a planning problem $\mathbf{P} = (S_0, \mathcal{O}, G)$ and an integer k encoded in binary, does there exist a plan in $\mathbf{P}$ of length k or less that achieves G?

3 Decidability and Undecidability Results

In this section, we show that logic programming is essentially the same as planning without delete lists. This is established by showing how to transform a deletion-free planning domain into a logic program such that for all goals G, the goal G is achievable from the planning domain iff the logical query that G represents is provable from the corresponding logic program.[5] Furthermore, we show that every logic program may be transformed to an equivalent planning domain. As a consequence of these equivalences, we can use results on the complexity of logic programs and deductive databases to demonstrate the following results:

- In the presence of function symbols, PLAN EXISTENCE is *undecidable* even if:
  - we have no delete lists;
  - we have no delete lists, and all operators have a most one positive precondition and no negative preconditions.

  The presence or absence of negative preconditions in the planning operators makes no difference where decidability is concerned[6] (though, as we shall see later, it does make a difference when we study the complexity of decidable planning domains).

[4]This definition follows the standard procedure for converting optimization problems into yes/no decision problems. What really interests us, of course, is the problem of finding the shortest plan that achieves G. This problem is at least as difficult as PLAN LENGTH, and in some cases harder. For example, in the Towers of Hanoi problem [1] and certain generalizations of it [17], the length of the shortest plan can be found in low-order polynomial time—but actually producing this plan requires exponential time and space, since the plan has exponential length. For more information on the relation between the complexity of optimization problems and the corresponding decision problems, see [16, pp. 115–117].

[5]This should be intuitively true, anyway, but the formal establishment of this equivalence is necessary before attempting to apply results from logic programming and deductive databases to planning problems. An important point to note is that we will only be considering truth in *Herbrand* models (cf. Shoenfield [35]) in this paper. Our undecidability/decidability results rely on this fact. In the context of our domain representation, it doesn't make much sense to consider non-Herbrand models because the domains of such models often contain objects that do not occur in the language, and these objects can not be referred to indirectly either, as we do not allow universal quantification. In the case of blocks world, for instance, this corresponds to assuming (inside the model) that there are blocks on the table that cannot be referred to in the language. Obviously, this is not relevant to planning. Thus, when we talk of logical consequences of programs, we will be referring to those sentences that are true in all Herbrand models of the program. For function-free languages, this condition is well known to yield decidability of logical consequence [32, 39].

[6]The only exception is when $\mathcal{L}$ is extended to contain infinitely many constants. This is discussed in detail in section 4.1

- PLAN EXISTENCE is *decidable* if we do not allow function symbols in our language and our first-order language is finitely generated (in particular, this means that only finitely many ground terms are present in the language). The presence or absence of delete lists does not affect the decidability result.

- In the presence of function symbols, PLAN EXISTENCE is *decidable* for positive, deletion-free planning domains that possess certain acyclicity and bounded-ness properties. Acyclicity properties are defined in terms of the syntactic dependencies between different operators in the planning domain, and are independent of the initial state.

- When our planning domain $\mathbf{P} = (S_0, \mathcal{O})$ is fixed in advance, then the problem "given a goal G, does there exist a plan that achieves G?" may still be undecidable depending on $\mathbf{P}$.[7] The presence or absence of delete lists does not affect this result.

- Even if we extend the operator definition to allow conditional effects, all of the above results still hold.

3.1 Equivalence between Logic Programming and Planning

We now proceed to establish the equivalence between logic programming and planning without delete lists. Subsequently (in Section 3), we show how to do away with delete lists when function symbols are absent.

If $\mathbf{P}$ is deletion-free, then the *logic program translation* of an operator $\alpha \in \mathcal{O}$, denoted by $\mathrm{LP}(\alpha)$, is the set of clauses

$$\mathrm{LP}(\alpha) = \{(\forall)(A \leftarrow B_1 \,\&\, \ldots \,\&\, B_n) : A \in \mathrm{Add}(\alpha)\},$$

where $\mathrm{Pre}(\alpha) = \{B_1, \ldots, B_n\}$.

Definition 3.1 The *logic program translation* of a planning domain $\mathbf{P} = (S_0, \mathcal{O})$, denoted $\mathrm{LP}(\mathbf{P})$, is the set of clauses

$$\mathrm{LP}(\mathbf{P}) = S_0 \cup \bigcup_{\alpha \in \mathcal{O}} \mathrm{LP}(\alpha).$$

Remark 3.1 Note that if we consider planning domains $\mathbf{P} = (S_0, \mathcal{O})$ where S_0 is infinite, then $\mathrm{LP}(\mathbf{P})$ would contain infinitely many unit clauses. The infinite nature of $\mathrm{LP}(\mathbf{P})$ will turn out to be irrelevant as far as establishing the equivalences between planning and logic programming are concerned (cf. Theorems 3.1 and 3.2) and the undecidability results that follow from the equivalence (cf. Theorem 3.8). This irrelevance is due to the compactness theorem for first-order logic.

Note that if $\mathbf{P} = (S_0, \mathcal{O})$ is a positive deletion-free planning domain, then $\mathrm{LP}(\mathbf{P})$ is a *definite* (i.e., negation-free) logic program. The following theorem shows that achievability of a goal G in $\mathbf{P}$ is identical to provability of G from $\mathrm{LP}(\mathbf{P})$.

[7] The phrase "$\mathbf{P} = (S_0, \mathcal{O})$ is fixed in advance" means that the planning domain $\mathbf{P}$ is not part of the input to a Turing machine; the only input is the goal G. In other words, $\mathbf{P}$ is a *specific* planning domain, and hence, there are Turing machines that can (intuitively) be specialized for the purpose of generating plans in this specific domain. In many well known planning problems, the set of operators is fixed; for example, in the blocks world (see Example 5.1), we have only four operators: stack, unstack, pickup and putdown. Our result shows that even if the set of operators is fixed, then depending on what the operators are, it still may be undecidable whether or not there is a plan for G.

Theorem 3.1 (Equivalence Theorem I) *Suppose* $\mathbf{P} = (S_0, \mathcal{O})$ *is a positive, deletion-free planning domain and* G *is a goal. Then there is a plan to achieve* G *from* $\mathbf{P}$ *iff* $\mathrm{LP}(\mathbf{P}) \models G$.

Definition 3.2 Suppose C is a definite Horn clause, i.e. a universally closed statement of the form

$$A \;\leftarrow\; B_1 \,\&\ldots\&\, B_n$$

where $A, B_1, \ldots, B_n$ are all atoms. When $n = 0$, C is said to be a *fact*. The *planning operator associated with* C, denoted op_C, is specified as follows:

Name:	$op_C(\vec{X})$,	where $\vec{X}$ is a vector of all variables occuring in C
Pre:	$\{B_1, \ldots, B_n\}$	
Add:	$\{A\}$	
Del:	$\emptyset$.	

Given a definite logic program P, the *planning domain translation* of P, denoted $\mathrm{PD}(P)$, is the pair $(S(P), \mathcal{O}(P))$, where

$$\begin{aligned} S(P) &= \{A : A \text{ is a ground instance of a fact in } P\}; \\ \mathcal{O}(P) &= \{op_C : C \text{ is a clause in } P,\ C \text{ not a fact}\}. \end{aligned}$$

Theorem 3.2 (Equivalence Theorem II) *Suppose* P *is a definite logic program and* G *is any goal. Then* $P \models G$ *iff there is a plan to achieve* G *from* $\mathrm{PD}(P) = (S(P), \mathcal{O}(P))$.

Theorem 3.1 holds only when $\mathbf{P}$ is positive. The reason for this is that if $\mathbf{P}$ is not positive, then $\mathrm{LP}(\mathbf{P})$ is a logic program that may contain negation in its body. Logic programming interprets negation in $\mathrm{LP}(\mathbf{P})$ as "failure to prove", which is different than the interpretation of negation in the planning domain $\mathbf{P}$. Intuitively, negation in logic programming says "conclude $\neg p$ if it is impossible to prove p". The corresponding notion of negation in planning would be "conclude $\neg p$ if there is no plan to achieve p" which is much stronger than saying "p is false in the current state." Thus, if $\mathbf{P}$ is not positive, then in some cases G will be achievable in $\mathbf{P}$ but $\mathrm{LP}(\mathbf{P}) \models G$ will be false. To see this, consider the following example:

Example 3.1 Consider the planning domain $\mathbf{P} = (S_0, \mathcal{O})$ that contains the following two operators α_1, α_2:

$$\begin{aligned} \mathrm{Pre}(\alpha_1) &= \{\neg b\} \qquad & \mathrm{Pre}(\alpha_2) &= \{c\} \\ \mathrm{Add}(\alpha_1) &= \{a\} \qquad & \mathrm{Add}(\alpha_2) &= \{b\} \end{aligned}$$

Suppose our initial state is the state $\{c\}$. Clearly, there is a plan to achieve a by simply executing operation α_1 in the initial state.

Now consider $\mathrm{LP}(\mathbf{P})$, which is the following logic program:

$$\begin{aligned} &a \leftarrow \neg b \\ &b \leftarrow c \\ &c \leftarrow \end{aligned}$$

The set of atoms provable from this program according to logic programming (all major semantics for logic programs agree on this program) is $\{b, c\}$, i.e. a cannot be obtained even though our planning domain admits a plan that achieves a.

3.2 Undecidability and Decidability

By combining Theorem 3.1 and Theorem 3.2 with various known decidability and undecidability results about logic programs, we obtain the following results. For our purposes, the most important of these results is Corollary 3.3, which says that even if we place some rather strict restrictions on the nature of the planning operators, PLAN EXISTENCE is undecidable.

At this point we should emphasize that function symbols are allowed, unless explicitly stated otherwise.

Corollary 3.1 (Semi-Decidability Results)

1. *$\{G : G$ is an existential goal such that there is a plan to achieve G from $\mathbf{P} = (S_0, \mathcal{O})\}$ is a recursively enumerable subset of the set of all goals.*

2. *Given any recursively enumerable collection X of ground atoms (which, of course, are goals), there is a positive deletion-free planning domain $\mathbf{P} = (S_0, \mathcal{O})$ such that $\{A : A$ is a ground atom such that there is a plan to achieve A from $\mathbf{P}\} = X$*

3. *If we restrict $\mathbf{P}$ to be positive and deletion-free, then* PLAN EXISTENCE *is strictly semi-decidable.*

Corollary 3.2 *The problem "given a positive deletion-free planning domain $\mathbf{P} = (S_0, \mathcal{O})$, is the set of goals achievable from $\mathbf{P}$ decidable?" is Π^0_2-complete.*

Corollary 3.3 *If we restrict $\mathbf{P}$ to be positive, deletion-free, and context-free, then* PLAN EXISTENCE *is still strictly semi-decidable.*

Corollary 3.4 *Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a positive, deletion-free planning domain. Then the problem: "given a goal G, does there exist a plan to achieve G?" is decidable iff the set of goals provable from* $\mathrm{LP}(\mathbf{P})$ *is decidable.*

In the above undecidability results, one restriction we did *not* make was to disallow function symbols in the planning language. In this section, we show that if function symbols are not allowed, then planning is decidable. To do this, we first prove decidability in a restricted case, where the planning operators are also deletion-free and function-free:

Theorem 3.3 *If we restrict $\mathbf{P}$ to be deletion-free and function-free, then* PLAN EXISTENCE *is decidable.*

We now show that when $\mathcal{L}$ contains no function symbols, we can do away with delete lists. The idea is intuitively the same as that of Green [18, 28] (vis-a-vis the famous "Green's formulation of planning"), with one difference: Green introduces function symbols even if the original language contained none: we introduce new constants. When the language is function-free, only finitely many new constants are included.

Theorem 3.4 (Eliminating Delete Lists and Negated Preconditions) *Suppose $\mathbf{P}$ is a function-free planning domain. Then there is a positive deletion-free planning domain $\mathbf{P}' = (S'_0, \mathcal{O}')$ such that for each goal*

$$G \equiv (\exists)(A_1 \,\&\, \ldots \,\&\, A_n),$$

there is a goal

$$G' \equiv (\exists)(A'_1 \,\&\ldots\&\, A'_n \,\&\, \text{poss}(S)),$$

*where "*poss*" is a new unary predicate symbol and for all* $1 \leq i \leq n$*, if* $A_i \equiv p(t_1, \ldots, t_n)$*, then* $A'_i \equiv p(t_1, \ldots, t_n, S)$ *where* S *is a variable symbol. Furthermore,* G *is achievable from* $\mathbf{P}$ *iff* G' *is achievable from* $\mathbf{P}'$*.*

An important point to note is that even though delete lists may be removed, the size of $\mathbf{P}'$ is much larger than $\mathbf{P}$, and our reduction is by no means polynomial. It is possible, but very unlikely that such a polynomial reduction exists, because we have proved that plan existence without any restrictions is EXPSPACE-complete, where as the same problem for positive deletion-free domains is EXPTIME-complete. A polynomial reduction from the unrestricted case to the other would imply EXPTIME = EXPSPACE, which would be a very significant result indeed. Whether EXPSPACE = EXPTIME or not, has been one of the most difficult questions of theory of computing. Although most researchers believe it is false, nobody has been able to come up with a proof of it (so far).

Theorem 3.4 allows us to establish the decidability of plan existence in function-free domains as follows: given a function-free planning domain $\mathbf{P}$, convert it into a function-free, deletion-free planning domain $\mathbf{P}'$ by using the transformation procedure in the proof of Theorem 3.4 (which is contained in the appendix). Theorem 3.3 then allows us to conclude that plan existence in the function-free, deletion-free domain $\mathbf{P}'$ is decidable. As exactly the same goals are achievable in the domains $\mathbf{P}'$ and $\mathbf{P}$, it follows that plan existence in $\mathbf{P}$ is decidable. This summarizes the proof of the following result.

Theorem 3.5 (Decidability of Function-Free Planning) *If we restrict* $\mathbf{P}$ *to be function-free, then* PLAN EXISTENCE *is decidable.*

3.3 Restricted Planning Domains

Though Corollary 3.1 indicates that planning in the presence of function symbols is undecidable, we can place specific restrictions on planning domains (even in the presence of function symbols) that guarantee decidability. In Section 3.3.1, we introduce certain syntactic acyclicity properties, and in Section 3.3.2, we introduce two semantic properties: one defines a class of planning domains (called weakly recurrent domains), while the other characterizes a class of goals called bounded goals. Any planning domain that is acyclic turns out to be weakly recurrent as well. We then use known results on weakly recurrent logic programs [3] to derive decidability results for weakly recurrent planning domains.

3.3.1 Acyclic Planning Domains

Definition 3.3 A *level mapping* for a language L is a mapping $\ell : AT(L) \rightarrow \mathbf{N}$ where $AT(L)$ is the set of ground atoms in language L and $\mathbf{N}$ is the set of natural numbers.
A *predicate level mapping* for a language L is a mapping $\sharp : Pred(L) \rightarrow \mathbf{N}$ where $Pred(L)$ is the set of predicate symbols in language L.

Definition 3.4 A logic program P is said to be *atomically acyclic* iff there exists a level mapping ℓ such that whenever

$$A \;\leftarrow\; B_1 \,\&\ldots\&\, B_n$$

is a ground instance of a clause in P, $\ell(A) > \ell(B_i)$ for all $1 \leq i \leq n$. ℓ is said to be a *witness* to the atomic acyclicity of P.

Analogously, a logic program P is said to be *predicate acyclic* there exists a *predicate level mapping* $\sharp$ such that whenever

$$p_0(\vec{t_0}) \quad \leftarrow \quad p_1(\vec{t_1})\,\&\ldots\&\,p_n(\vec{t_n})$$

is a clause (not necessarily ground) in P, $\sharp(p_0) > \sharp(p_i)$ for all $1 \leq i \leq n$.

Intuitively, atomic acyclicity guarantees that the dependencies expressed in the program are not recursive. For example, the program containing clauses

$$\begin{array}{rcl} p & \leftarrow & q \\ q & \leftarrow & p \end{array}$$

involves a cycle. This program is not acyclic.

It is also easy to see that predicate acyclicity implies atomic acyclicity – to see this, note that if $\sharp$ is a witness to the predicate acyclicity of P, then we can define a level mapping ℓ as follows:

$$\ell(p(\vec{t})) \quad = \quad \sharp(p).$$

It is straightforward to verify that ℓ is a witness to the atomic-acyclicity of P.

Furthermore, observe that predicate-acyclicity can be checked in linear-time. To see this, observe that given a logic program P, we can draw a graph on the predicate symbols in P as follows: there is an arc from p to q iff there is a clause in P having an atom of the form $p(\vec{t})$ in its head, and an atom of the form $q(\vec{s})$ in its body. This graph can be constructed in linear time by reading in clauses in P one by one. P is acyclic iff the resulting graph is acyclic. The result follows as checking for cycles in a graph can be solved in linear time.

We now show how the definitions of predicate acyclicity and atomic acyclicity can be extended to apply to positive, deletion-free planning domains.

Definition 3.5 Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a positive, deletion-free planning domain. $\mathbf{P}$ is said to be *atomically acyclic* iff there exists a level mapping ℓ such that for all ground instances, α, of operators in $\mathbf{P}$, it is the case that $\ell(A) > \ell(B)$ for all $A \in Add(\alpha)$ and $B \in Pre(\alpha)$.

Definition 3.6 Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a positive, deletion-free planning domain. $\mathbf{P}$ is said to be *predicate acyclic* iff there exists a predicate level mapping $\sharp$ such that for all operators α in $\mathbf{P}$, it is the case that $\sharp(p) > \sharp(q)$ for all predicates p occurring in $Add(\alpha)$ and all predicates q occurring in $Pre(\alpha)$.

As in the case of logic programs, if a planning domain is predicate acyclic then it is also atomically acyclic. Furthermore, the following two results shows a close connection between acyclicity of logic programs and planning domains.

Proposition 3.1 *Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a positive, deletion-free planning domain. Then:*

1. *If $\mathbf{P}$ is atomically acyclic, then the logic program translation,* LP($\mathbf{P}$)*, of $\mathbf{P}$, is atomically acyclic.*

2. *If $\mathbf{P}$ is predicate acyclic, then* LP($\mathbf{P}$) *is predicate acyclic.*

The above proposition is true because the level mapping that witnesses the predicate/atomic acyclicity of $\mathbf{P}$ also witnesses the predicate/atomic acyclicity of LP($\mathbf{P}$). The same reasoning may be applied to prove the next result.

Proposition 3.2 *Suppose P is a definite logic program. Then:*

1. *If P is atomically acyclic, then* PD(P) *is atomically acyclic.*

2. *If P is predicate acyclic, then* PD(P) *is predicate acyclic.*

3.3.2 Weakly Recurrent Planning Domains

The above two results show that our transformations, LP, and PD, from logic programs to planning domains (and vice-versa) preserve atomic and predicate acyclicity. Bezem [3] has shown that a class of logic programs called *weakly recurrent* programs possess appealing decidability properties. All predicate and atomically acyclic programs are weakly recurrent (though the converse may not be true). Below, we will present Bezem's definition of weakly recurrent logic programs, and then show how to define an analogous notion for planning domains which allows these decidability properties to be applied to planning.

Definition 3.7 (Bezem [3]) A definite logic program is *weakly recurrent* iff there exists a level mapping ℓ such that for every clause in P having a ground instance of the form:

$$A \quad \leftarrow \quad B_1 \,\&\ldots\&\, B_n$$

such that if $P \not\models A$ (i.e. A is not a logical consequence of P), there exists an $1 \leq i \leq n$ such that $P \not\models B_i$ and $\ell A) > \ell(B_i)$.

Intuitively, a weak recurrence says that non-provability of A can be established by verifying the non-provability of some strictly lower-level atoms B.

Theorem 3.6 (Bezem [3]) *If P is a weakly recurrent definite logic program, then the set of ground atoms provable from P is recursive (i.e. decidable).*
Furthermore, every recursive set of ground atoms of language L can be expressed as the set of ground atoms provable from some weakly recurrent logic program P.

Note that the second part of the above theorem does not say that every program whose set of ground atomic consequences is recursive is weakly-recurrent. There are non-recurrent programs with a recursive set of ground atomic consequences. We now show how the notion of weakly recurrent logic programs can be used to define a similar notion for planning domains.

Definition 3.8 A planning domain $\mathbf{P} = (S_0, \mathcal{O})$ is *weakly recurrent* iff there exists a level mapping ℓ such that for every ground instance, α, of an operator in $\mathcal{O}$, it is the case that:

> If $A \in Add(\alpha)$ is such that there is no plan to achieve A from $\mathbf{P}$, then there is a $B_i \in Pre(\alpha)$ such that there is no plan to achieve B_i from $\mathbf{P}$ and $\ell(A) > \ell(B_i)$.

The property of weak recurrence is preserved by the transformations PD and LP.

Proposition 3.3 *(1) If* $\mathbf{P} = (S_0, \mathcal{O})$ *is a weakly recurrent, positive, deletion-free planning domain, then* LP($\mathbf{P}$) *is a weakly recurrent logic program.*
(2) Conversely, if P is a weakly recurrent definite logic program, then $PD(P)$ is a weakly recurrent planning domain.

Part (1) of Proposition 3.3 follows immediately because the level mapping that witnesses the weakly-recurrent property of $\mathbf{P}$ also witnesses the weakly-recurrent property of LP($\mathbf{P}$). Part (2) follows similarly.

Definition 3.9 Suppose $G = (\exists)(A_1 \& \ldots \& A_n)$ is a goal. Let $Grd(G)$ denote the set of all ground instances of the quantifier-free conjunction $(A_1 \& \ldots \& A_n)$. G is said to be *bounded* w.r.t. a level mapping ℓ iff there exists an integer b such that for every ground instance $(A_1 \& \ldots \& A_n)\theta$ in $grd(G)$, it is the case that

$$\ell(A_i) < b.$$

Note, in particular, that when our language L is function-free, all goals are bounded w.r.t. any level mapping.

Example 3.2 Consider the language L containing one unary predicate symbol p one unary function symbol s, and one constant symbol a. Let ℓ be the level mapping which assigns 0 to $p(a)$, 1 to $p(s(a))$, 2 to $p(s(s(a)))$, and i to $p(s^i(a))$. Then the goal $(\exists X)p(X)$ is not bounded w.r.t. level mapping ℓ. On the other hand, if we consider the level mapping ℓ' that assigns 0 to $p(a), p(s(s(a)), \ldots, p(s^{2i}(a)), \ldots,$ and 1 to $p(s(a)), p(s(s(s(a)))), \ldots, p(s^{2i+1}(a)), \ldots,$ then $(\exists X)p(X)$ is bounded w.r.t. level mapping ℓ'.

Theorem 3.7 (Decidability for Weakly Recurrent Planning Domains) *If* $\mathbf{P} = (S_0, \mathcal{O})$ *is restricted to be weakly recurrent (via witness ℓ), positive, and deletion-free, and G is restricted to be bounded w.r.t. ℓ, then* PLAN EXISTENCE *is decidable.*

As

$$\text{Predicate Acyclic} \Longrightarrow \text{Atomically Acyclic} \Longrightarrow \text{Weakly Recurrent},$$

the following results are immediate.

Corollary 3.5 *If* $\mathbf{P}$ *is a (predicate, resp. atomically) acyclic planning domain, and G is a bounded goal (w.r.t. the level mapping that establishes* $\mathbf{P}$*'s acyclicity), then the problem: "Is G achievable from* $\mathbf{P}$ *?" is decidable.*

Furthermore, for all predicate acyclic (via $\sharp$) planning domains, every goal G is bounded—take the bound b to be

$$1 + \max \{\sharp(p) \mid p \text{ is a predicate symbol in language } L\}$$

Hence:

Corollary 3.6 *If* $\mathbf{P}$ *is a predicate acyclic planning domain, and G is any goal, then the problem: "Is G achievable from* $\mathbf{P}$ *?" is decidable.*

3.4 Extended Planning Domains

Earlier, we defined the language $\mathcal{L}$ to contain only finitely many constant symbols, and states in $\mathcal{L}$ to contain only finitely many ground atoms. In this section, we consider what happens when one or both of these assumptions is violated. We also consider what happens if the planning operators are extended to allow conditional effects.

3.4.1 Infinite Initial States; Infinitely Many Constants

Corollary 3.1 showed that PLAN EXISTENCE is semi-decidable even if $\mathbf{P}$ is positive and deletion-free. One might think that allowing infinite initial states would increase the difficulty of PLAN EXISTENCE even further, but the following theorem shows that it does not.

Theorem 3.8 *If $\mathbf{P} = (S_0, \mathcal{O})$ is restricted to be positive and deletion-free, but the initial state S_0 is allowed to be an infinite, decidable set of ground atoms,* PLAN EXISTENCE *is (strictly) semi-decidable.*

The following theorem states that if the initial state is allowed to be infinite and the number of constant symbols in the language is allowed to be infinite, then any planning domain whose language contains function symbols can be reduced to an equivalent planning domain whose language contains no function symbols.

Theorem 3.9 *Let $\mathbf{P} = (S_0, \mathcal{O})$ be any planning domain whose language $\mathcal{L}$ contains function symbols, and whose initial state S_0 may possibly be infinite. Then there is a planning domain $\mathbf{P}' = (S_0', \mathcal{O}')$ having the following properties: $\mathbf{P}'$'s language $\mathcal{L}'$ contains infinitely many constants but no function symbols, the initial state S_0' may be infinite, and for every goal G in $\mathbf{P}$, there is a goal G' in $\mathbf{P}'$ such that there is a plan for G in $\mathbf{P}$ iff there is a plan for G' in $\mathbf{P}'$.*

In the above theorem, the basic idea is to encode each term $f(t_1, \ldots, t_n)$ of $\mathcal{L}$ as a constant symbol $k_{f(t_1,\ldots,t_n)}$ in $\mathcal{L}'$. To do this, we must also add an "equivalence condition", i.e., an atom saying that applying f to $t_1, \ldots, t_n$ yields $k_{f(t_1,\ldots,t_n)}$. This atom must appear in the preconditions of every planning operator that contains the term $f(t_1, \ldots, t_n)$, and thus it must also appear in every state. Furthermore, if any of the terms $t_1, \ldots, t_n$ contains function symbols, then it must be encoded in the same way.

Since the above encoding adds no deletions and no negative preconditions to the operators, the following corollary follows immediately:

Corollary 3.7 *If $\mathbf{P}$ is positive and/or deletion-free, then $\mathbf{P}'$ is too.*

From Corollary 3.3, if the language contains function symbols then planning is undecidable, even if the planning domain is restricted to be positive and deletion-free. Thus, from the above results, the following corollary follows immediately:

Corollary 3.8 *If we restrict $\mathbf{P} = (S_0, \mathcal{O})$ to be positive, deletion-free, and function-free, but allow S_0 to be infinite and allow $\mathbf{P}$'s language to contain infinitely many constant symbols, then* PLAN EXISTENCE *is semi-decidable.*

Corollary 3.8 subsumes Theorem 3.8, in the following sense. Although the theorem restricts the set of constant symbols to be finite, it allows function symbols. Any function-free planning domain that contains infinitely many constant symbols $c_1, c_2, \ldots$ can easily be mapped into an equivalent planning domain that contains one constant symbol c and one unary function symbol f, by mapping $c_1 \mapsto c$, $c_2 \mapsto f(c)$, and so forth. Thus, Corollary 3.8 shows that even with further restrictions than described in Theorem 3.8, planning is still undecidable.

One might think that if the planning language is allowed to contain infinitely many constants, this should be sufficient to make PLAN EXISTENCE undecidable even if the initial state is restricted to be finite. However, decidability depends on whether all of these constants are relevant for planning. If the initial state is finite, then all but a finite number of ground atoms will be false in the initial state. An operator can introduce a new constant to a plan only if that constant does not appear in any of the operator's positive preconditions. Thus, when we restrict the domain to be positive and deletion-free, the only way we can introduce a new constant is by using an operator with no preconditions. However, in this case there is no reason why we should not use a *basic constant* (i.e., a constant that appears in the initial state or in an operator definition) to do the same job. Hence the problem is decidable. On the other hand, if we allow negated preconditions, the previous argument does not hold. We can introduce new constants by using operators with negated preconditions, and we cannot replace these constants with basic constants. Hence the problem becomes undecidable.

The above argument leads to the following theorems:

Theorem 3.10 *If* $\mathbf{P}$*'s language* $\mathcal{L}$ *is allowed to contain infinitely many constants, then* PLAN EXISTENCE *is semi-decidable even if* $\mathbf{P} = (S_0, \mathcal{O})$ *is restricted to be deletion-free and function-free (and* S_0 *is finite).*[8]

The statements of Theorem 3.10 and Corollary 3.8 are quite similar. In both of them, we have extended the language to allow infinitely many constant symbols, and have restricted the operators to be deletion-free and function-free. Under these conditions, PLAN EXISTENCE is undecidable if we either

- allow the initial state to be infinite (Corollary 3.8), or
- allow non-positive operators (Theorem 3.10).

Under this same set of conditions, the following theorem says that if we restrict the initial state to be finite *and* the operators to be positive, then PLAN EXISTENCE is decidable.

Theorem 3.11 *If the language is allowed to contain infinitely many constants but* $\mathbf{P} = (S_0, \mathcal{O})$ *is restricted to be positive, deletion-free, and function-free (and* S_0 *is finite), then* PLAN EXISTENCE *is decidable.*

3.4.2 Conditional Operators

Several researchers [8, 10, 31, 30] have been interested in actions whose effects are context dependent, that is, dependent on the input situation. Thus, we found it necessary to examine the complexity of planning with such operators. We will be using Dean's [10] formulation of these operators, which is a more general version of what Chapman [8] uses.

[8]Tom Bylander (personal communication) has proved a more restricted version of this theorem, in which he requires that the planning operators be allowed to contain delete lists. In proving our more general theorem, we have benefited from his proof technique.

Definition 3.10 A *conditional operator* α is a finite set $\{t_1, t_2, \ldots, t_n\}$, where each t_i is a triple of the form $\langle \mathrm{Pre}_i, \mathrm{Del}_i, \mathrm{Add}_i \rangle$. Pre_i, Del_i, and Add_i correspond to the precondition list, delete list and add list associated with the i'th triple, respectively. Hence each of these lists are sets of atoms.

Definition 3.11 Let α be a conditional operator, θ be a ground substitution for the variables appearing in α, S be a state, and

$$S' = (S - \bigcup_{i \in I} \mathrm{Del}_i\theta) \cup \bigcup_{i \in I} \mathrm{Add}_i\theta,$$

where

$$I = \{i : S \text{ satisfies } \mathrm{Pre}_i\theta\}.$$

Then we say that α is *θ-executable* in state S, *resulting* in state S'. This is denoted as

$$S \overset{\alpha,\theta}{\Longrightarrow} S'.$$

The definitions of *positive* and *deletion-free* can be trivially extended to include conditional operators.

Obviously, planning with regular operators is a special case of planning with conditional operators, where each conditional operator is restricted to contain exactly one triple $\langle \mathrm{Pre}, \mathrm{Del}, \mathrm{Add} \rangle$. Thus planning with conditional operators is at least as hard as planning with regular operators, so all of our undecidability results still hold if conditional operators are allowed.

The next point to be investigated is whether allowing conditioal operators affects our decidability results (Theorems 3.3, 3.5, 3.7, and 3.11). Below, we show that it does not:

- Theorems 3.3 and 3.5 restrict the planning domain to be function-free. Thus we have only a finite number of ground terms, so the number of states is finite. Hence, we can search all the states reachable from the initial state to see whether one of them satisfies the goal in finite time, so the problem remains decidable. Thus Theorems 3.3 and 3.5 are unaffected if conditional operators are allowed.

- Theorems 3.7 and 3.11 restrict the planning domain to be positive and deletion-free. If a positive, deletion-free planning domain contains conditional operators, the following transformation creates an equivalent positive and deletion-free planning domain that has no conditional operators. This shows that allowing conditional operators does not affect Theorems 3.7 and 3.11.

  Let $\mathbf{P} = (S_0, \mathcal{O})$ be a planning domain in which the operator definition is extended to allow conditional operators. Let $\alpha = \{t_1, \ldots, t_n\}$ be any one of these conditional operators, where for each i, $t_i = \langle \mathrm{Pre}_i, \mathrm{Del}_i, \mathrm{Add}_i \rangle$.

  We can define an equivalent set of STRIPS-style operators, none of which has conditional effects. For every subset $I \subseteq \{1, \ldots, n\}$, α_I is the following STRIPS-style operator:

Name:	$\alpha_I(V_I)$
Pre:	$\bigcup_{i \in I} \mathrm{Pre}_i$
Del:	$\bigcup_{i \in I} \mathrm{Del}_i$
Add:	$\bigcup_{i \in I} \mathrm{Add}_i$

where V_I consists of all variables appearing in Pre(α_I), Del(α_I), and Add(α_I).

Suppose we are given a planning domain $\mathbf{P}$ that contains the conditional operator $\alpha = \{t_1, \ldots, t_n\}$, and let $\mathbf{P}'$ be the planning domain in which α is replaced by the set of unconditional operators $\{\alpha_I\}$ defined above. Then $\mathbf{P}$ and $\mathbf{P}'$ are equivalent planning domains, in the sense that $S \overset{\alpha,\theta}{\Longrightarrow} S'$ in $\mathbf{P}$ if and only if there is an operator α_I in $\mathbf{P}'$ that is θ-executable in S, and $S \overset{\alpha_i,\theta}{\Longrightarrow} S'$.

The above arguments prove the following:

Proposition 3.4 (Irrelevance of Conditional Operators for Decidability) *Whether or not the definition of a planning domain is extended to allow conditional planning operators makes no difference in any of our decidability and undecidability results.*

4 Comparison with Chapman's Undecidability Results

To date, the best-known results on decidability and undecidability in planning systems are those of Chapman [8]. However, there is a certain amount of confusion about what Chapman's undecidability results actually say, because some of his assumptions become clear only after a careful reading of the paper. To clarify the meaning of Chapman's undecidability results, we now compare and contrast his results with ours.

4.1 First Undecidability Theorem

Chapman's first undecidability theorem ([8, pp. 370–371]) says that all Turing machines with their inputs may be encoded as planning problems in the TWEAK representation, and hence planning is undecidable. To prove this theorem, Chapman makes use of the following assumptions:

1. the planning language is function-free;
2. "an infinite [but recursive] set of constants t_i are used to represent the tape squares" [8, p. 371];
3. the initial state is infinite (but recursive). In particular, "there must be countably many successor propositions to encode the topology of the tape (and also countably many contents propositions to make all but finitely many squares blank)" [8, p. 371].

Our Corollary 3.8 subsumes this result, by showing that with the same set of assumptions, PLAN EXISTENCE is undecidable even if all of the planning operators are positive and deletion-free.

In his discussion of the First Undecidability Theorem [8, p. 344], Chapman says:

> This result is weaker than it may appear ... the proof uses an infinite (though recursive) initial state to model the connectivity of the tape. It may be that if problems are restricted to have finite initial states, planning is decidable. (This is not obviously true though. There are infinitely many constants, and an action can in effect "gensym" one by referring to a variable in its post-conditions that is not mentioned in its preconditions.)

Our Theorems 3.10 and 3.11 solve the open problem posed in the above quote. In particular, suppose that Chapman's first two assumptions are satisfied (i.e., the language is function-free, and there are infinitely many constant symbols), but the initial state is finite. Then:

- PLAN EXISTENCE is undecidable, even if all operators are deletion-free;
- if the operators are both deletion-free and positive, then PLAN EXISTENCE is decidable.

4.2 Second Undecidability Theorem

The statement of Chapman's second undecidability theorem is that "planning is undecidable even with a finite initial state if the action representation is extended to represent actions whose effects are a function of their input situations" [8, p. 373].

The meaning of the phrase "effects are a function of their input situations" has caused some confusion. Several researchers, including ourselves [11] and Mark Peot (in his conference presentation of [31]), thought that Chapman meant a special case of the conditional operators defined in Section 3.4.2. However, our Proposition 3.4 shows that whether or not such operators are allowed makes no difference in the decidability of PLAN EXISTENCE—and an examination the proof of Chapman's theorem makes it clear he is referring to a different kind of operator.

In Chapman's proof of the theorem, he makes use of operators that increment and decrement two counters. Since there is no upper bound on the value of those counters, to define such operators formally would require the use of function symbols. Thus, his phrase "effects are a function of their input situations" apparently refers to operators that contain function symbols. Our Corollary 3.1 shows that if function symbols are allowed, then even if there are only finitely many constant symbols, then PLAN EXISTENCE is undecidable. Thus, Corollary 3.1 subsumes the Second Undecidability Theorem.

5 Complexity Results

As shown in Theorem 3.5, planning is decidable if our language contains finitely many constant symbols, and no function symbols. We now study the complexity of planning domains that satisfy these conditions. We discuss how delete lists, negated preconditions, propositional operators, and fixing the set of operators affects the complexity of planning.

5.1 Preliminaries for the Complexity Results

5.1.1 What is considered as Input?

Since the complexity of a problem is evaluated with respect to the length of the input, it is important to understand precisely what the input is. According to the definitions of PLAN EXISTENCE and PLAN LENGTH the problem input consists of a planning problem $\mathbf{P} = (S_0, \mathcal{O}, G)$, where

- S_0 is the initial state (a set of ground atoms);
- $\mathcal{O}$ is the set of available planning operators;
- G is the goal (an existentially closed set of atoms).

The planning language is the language $\mathcal{L}$ generated by the predicate symbols, function symbols, and constant symbols that appear in this input.

Unless we state otherwise, all complexity terms (polynomial, exponential, etc.), should be understood in terms of the length of the input, which we will denote by $||\mathbf{P}||$. In Section 5.3, we consider what happens if the set of operators is fixed, and thus excluded from the input—but we state this condition explicitly in each result that uses it.

5.1.2 Eliminating Negated Preconditions

In Theorem 3.4, we proved that delete lists and negated preconditions could be "compiled away," but this translation cannot be done in polynomial time. We show below that if we are willing to allow delete lists, then we can remove negations from preconditions of operators in polynomial time. Thus, if delete lists are allowed, then negated preconditions do not affect the complexity of planning.

Theorem 5.1 (Eliminating Negated Preconditions) *In polynomial time, given any planning domain $\mathbf{P} = (S_0, \mathcal{O})$ we can produce a positive planning domain $\mathbf{P}' = (S'_0, \mathcal{O}')$ having the following properties:*

1. *For every goal G, a plan exists for G in $\mathbf{P}$ if and only if a plan exists for G in $\mathbf{P}'$.*
2. *For every goal G and non-negative integer l, there exists a plan of length l for G in $\mathbf{P}$ if and only if there exists a plan of length $l + 2^{kv}$ for G in $\mathbf{P}'$, where k is the maximum arity among the predicates of $\mathbf{P}$ and $v = \lceil \lg c \rceil$, where c is the number of constants in $\mathbf{P}$ (i.e., v is the number of bits necessary to encode the constants in binary).*

To prove the above theorem, the basic idea is this:[9] for each predicate P in $\mathbf{P}$, we introduce another complementary predicate P' such that whenever P is true, P' is false. The operators in $\mathcal{O}$ can easily be modified to achieve this. The problem is that for every atom that is false in $\mathbf{P}$'s initial state S_0, we need to assert the corresponding complementary atom in $\mathbf{P}'$. Since there might be an exponential number of such atoms, we cannot just place them in S'_0. Instead, we assert all these atoms using operators, using a "counter" predicate to keep track of how many of them have been asserted. When all of these atoms have been asserted, we delete the ones corresponding to those appearing in S_0, assert the atoms of P that are in S_0, and set start() so that we can start imitating the behavior of the original planning problem.

Note that $\mathbf{P}'$ will not be deletion-free, even if $\mathbf{P}$ is.

5.2 Planning When the Operator Set is Part of the Input

In this section, we consider the complexity of planning in the "domain-independent" case, in which the operators are part of the input and thus different problem instances may have different operator sets.

5.2.1 Propositional Operators

The following theorems deal with the special case in which all predicates are propositions (i.e., 0-ary). In this case, the number of ground atoms in $\mathcal{L}$ is polynomial in $||\mathbf{P}||$, since

[9] We again remind the reader that complete proofs appear in the appendix.

each atom must appear somewhere in the input. Since a state may be any set of ground atoms, there is an exponential number of states. Since there are no variables, the number of operator instances is $|\mathcal{O}|$, which of course is polynomial in $||\mathbf{P}||$.

Theorem 5.2 (Bylander [5])

1. *If we restrict* $\mathbf{P}$ *to be propositional, then* PLAN EXISTENCE *is* PSPACE-*complete.*
2. *If we restrict* $\mathbf{P}$ *to be propositional and positive, then* PLAN EXISTENCE *is* PSPACE-*complete.*
3. *If we restrict* $\mathbf{P}$ *to be propositional and deletion-free, then* PLAN EXISTENCE *is* NP-*complete.*
4. *If we restrict* $\mathbf{P}$ *to be propositional, positive, and deletion-free, then* PLAN EXISTENCE *is in P.*
5. *If we restrict* $\mathbf{P}$ *to be propositional, positive, and side-effect-free, then* PLAN EXISTENCE *is in P.*

Synopsis of proof. Here are the basic intuitions behind the above theorem; for the details see [5]. In general, we might need to use the same operator instance more than once. For example, consider the propositional planning problem in which the initial state is $S_0 = \emptyset$, the goal is $G = \{p, q, r\}$, and the operators are

Name:	A	Name:	B
Pre:	$\emptyset$	Pre:	$\{p\}$
Del:	$\emptyset$	Del:	$\{q\}$
Add:	$\{p, q\}$	Add:	$\{r\}$

In order to achieve r, we need to use operator B. To satisfy the precondition p, we need to use operator A. However, since operator B deletes q, we need to use operator A a second time, to reassert q. Thus the plan is (A, B, A).

To handle such situations, we might have to search through all the states, using some operators more than once, doing an exponential amount of work—but since the size of each state is at most polynomial, we can do this search in PSPACE.[10]

If $\mathbf{P}$ is deletion-free, then once a proposition is asserted, it remains asserted throughout the plan. Thus, no operator needs to be used more than once, and the length of the plans are constrained to be polynomial. We still need to decide how to choose the operators and how to order them in the plan, and thus the problem is NP-complete.

If $\mathbf{P}$ is both positive and deletion-free, then no operator can clobber any goal nor any other operator, and any operator that is executable remains so throughout the plan. Thus, we no longer care which operators we choose, or how they are ordered. Instead, we can arbitrarily choose operators and apply them until either the goal is achieved, or no executable operator that has not yet been used remains. This takes polynomial time. ∎

If $\mathbf{P}$ is positive, deletion-free, and context-free, then we can do a backwards search for each proposition in the goal set. At each iteration, we nondeterministically choose an operator that achieves the subgoal, and we make its precondition the new subgoal. We

[10] Although it has not been proved, PSPACE is believed to be equal to EXPTIME.

repeat this until the subgoal is in the initial state, or no such operator to choose exists. If we can find a plan for each of the propositions in the goal, then these plans can be combined to make a plan for the goal. Since **P** is deletion free and positive, no operator can delete any of the preconditions of the other operators. We can do the backwards chaining because each operator has at most one precondition, and thus the number of subgoals do not increase. All these require logspace, and since we need to make non-deterministic choices, the problem is NLOGSPACE-complete. Thus, we have the following result, which is proved in the appendix.

Theorem 5.3 *If we restrict* **P** *to be propositional, positive, context-free, and deletion-free, then* PLAN EXISTENCE *is* NLOGSPACE-*complete.*

The following theorems and corollaries state our results on the complexity of PLAN LENGTH. Note that in several cases where PLAN EXISTENCE is in P (items 3 and 4 of Theorem 5.2, and Theorem 5.3), the corresponding PLAN LENGTH problem (Corollaries 5.2 and opty-1-cor, and Theorem 5.4, respectively) is NP-complete. The reason for this is as follows. For PLAN EXISTENCE, the restrictions allowed us to plan for each subgoal separately, using backwards chaining. We cannot do this for PLAN LENGTH, because of *enabling-condition interactions*. Enabling-condition interactions are discussed in more detail in [20], but the basic idea is that a sequence of actions that achieves one subgoal might also achieve other subgoals or make it easier to achieve them. Although such interactions will not affect PLAN EXISTENCE, they will affect PLAN LENGTH, because they make it possible to produce a shorter plan. It is not possible to detect and reason about these interactions if we plan for the subgoals independently; instead, we have to consider all possible operator choices and orderings, making PLAN LENGTH NP-hard.

Theorem 5.4 *If we restrict* **P** *to be propositional, positive, context-free and deletion-free, then* PLAN LENGTH *is* NP-*complete.*

Corollary 5.1 *If we restrict* **P** *to be propositional, positive and deletion-free, then* PLAN LENGTH *is* NP-*complete.*

Corollary 5.2 *If we restrict* **P** *to be propositional and deletion-free,* PLAN LENGTH *is* NP-*complete.*

If we allow non-empty delete lists, then we are no longer confined to plans of polynomial length, and thus the complexity of PLAN LENGTH increases, as stated in the following theorem.

Theorem 5.5 PLAN LENGTH *is* PSPACE-*complete if we restrict* **P** *to be propositional. It is still* PSPACE-*complete if we restrict* **P** *to be propositional and positive.*

5.2.2 Propositional Operators with Operator Composition

Both Theorem 5.3 and Clause 5 of Theorem 5.2 require restrictions on the number of clauses in the preconditions and/or postconditions of the planning operators. These restrictions can easily be weakened by allowing the operators to be composed, as described below.

Definition 5.1 An operator α is *composable* with another operator β if the positive preconditions of β and del(α) are disjoint, and the negative preconditions of β and add(α) are disjoint.

Definition 5.2 If α and β are composable, then the *composition* of α with β is

Pre: $\mathrm{Pre}(\alpha) \cup (P_1 - \mathrm{Add}(\alpha)) \cup (P_2 - del(\alpha))$
Add: $\mathrm{Add}(\beta) \cup (\mathrm{Add}(\alpha) - \mathrm{Del}(\beta))$
Del: $\mathrm{Del}(\beta) \cup (\mathrm{Del}(\alpha) - \mathrm{Add}(\beta))$

where P_1 and P_2, respectively, are the positive and negative preconditions of β.

Theorem 5.6 (Composition Theorem) *Let $\mathbf{P} = (S_0, \mathcal{O})$ be a planning domain, and $\mathcal{O}'$ be a set of operators such that each operator in $\mathcal{O}'$ is the composition of operators in $\mathcal{O}$. Then for any goal G, there is a plan to achieve G in $\mathbf{P}$ iff there is a plan to achieve G in $\mathbf{P}'$, where $\mathbf{P}' = (S_0, \mathcal{O} \cup \mathcal{O}')$.*

This theorem allows us to extend the scope of several of the complexity theorems.

Corollary 5.3 *Suppose we restrict $\mathbf{P} = (S_0, \mathcal{O}, G)$ to be such that $\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2$, where $\mathcal{O}_1$ is propositional, deletion-free, positive and context-free, and every operator in $\mathcal{O}_2$ is the composition of operators in $\mathcal{O}_1$. Then* PLAN EXISTENCE *is* NLOGSPACE*-complete.*

Proof. Immediate from Theorem 5.6 and Theorem 5.3. ∎

Corollary 5.4 *Suppose we restrict $\mathbf{P} = (S_0, \mathcal{O}, G)$ to be such that $\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2$, where $\mathcal{O}_1$ is propositional, positive, and side-effect-free, and every operator in $\mathcal{O}_2$ is the composition of operators in $\mathcal{O}_1$. Then* PLAN EXISTENCE *is in* P.

Proof. Immediate from Theorem 5.6 and Theorem 5.2. ∎

Example 5.1 (Blocks World) Bylander [5] reformulates the blocks world so that each operator is restricted to positive preconditions and one postcondition. Instead of the usual "on" and "clear" predicates, he uses proposition off_{ij} to denote that block i is not on block j. For each pair of blocks i and j, he has two operators: one that moves block i from the top of block j to the table, and one that moves block i from the table to the top of block j. These operators are defined as follows:

Name: $\mathrm{totable}_{ij}$
Pre: $\{\mathrm{off}_{1,i}, \mathrm{off}_{2,i}, \ldots, \mathrm{off}_{n,i}, \mathrm{off}_{1,j}, \mathrm{off}_{2,j}, \ldots, \mathrm{off}_{i-1,j}, \mathrm{off}_{i+1,j}, \ldots, \mathrm{off}_{n,j}\}$
Del: $\emptyset$
Add: $\{\mathrm{off}_{i,j}\}$

Name: $\mathrm{toblock}_{ij}$
Pre: $\{\mathrm{off}_{1,i}, \mathrm{off}_{2,i}, \ldots, \mathrm{off}_{n,i}, \mathrm{off}_{1,j}, \mathrm{off}_{2,j}, \ldots, \mathrm{off}_{n,j}, \mathrm{off}_{i,1}, \mathrm{off}_{i,2}, \ldots, \mathrm{off}_{i,n}\}$
Del: $\{\mathrm{off}_{i,j}\}$
Add: $\emptyset$

In Bylander's formulation of blocks world, $\mathbf{P}$ is positive and side-effect-free. Thus as a consequence of Clause 5 of Theorem 5.2, in Bylander's formulation of blocks world PLAN EXISTENCE can be solved in polynomial time.

In Bylander's formulation of the blocks world, it is not possible for blocks to be moved directly from one stack to another. This has two consequences, as described below.

The first consequence is that in Bylander's formulation of blocks world, PLAN LENGTH can be solved in polynomial time. To show this, below we describe how to compute how many times each block b must be moved in the optimal plan. Thus, to see whether or not there is a plan of length k or less, all that is needed is to compare k with

$$\sum_b \text{how many times } b \text{ must be moved.}$$

Let S be the current state, and b be any block. If the stack of blocks from b down to the table is consistent with the goal conditions (whether or not this is so can be determined in polynomial time [20]), then b need not be moved. Otherwise, there are three possibilities:

1. If b is on the table in S and the goal conditions require that b be on some other block c, then in the shortest plan, b must be moved exactly once: from the table to c.

2. If b is on some block c in S and the goal conditions require that b be on the table, then in the shortest plan, b must be moved exactly once: from c to the table.

3. If b on some block c in S and the goal conditions require that b be on some block d (which may be the same as c), then in the shortest plan, b must be moved exactly twice: from c to the table, and from the table to d.

The second consequence is that translating an ordinary blocks-world problem into Bylander's formulation will not always preserve the length of the optimal plan. The reason for this is that in the ordinary formulation of blocks world, the optimal plan will often involve moving blocks directly from one stack to another without first moving them to the table, and this cannot be done in Bylander's formulation. It appears that Bylander's formulation cannot be extended to allow this kind of move another without violating the restriction that each has only positive preconditions and one postcondition.

The above problem can easily be overcome by augmenting Bylander's formulation to include all possible compositions of pairs of his operators. Theorem 5.2 does not apply to this formulation, but Corollary 5.4 does apply, and gives the same result as before: PLAN EXISTENCE can be solved in polynomial time.

Since this extension to Bylander's formulation allows stack-to-stack moves, there is a one-to-one correspondence between plans in this formulation and the more usual formulations of the blocks world, such as those given in [7, 21, 28, 37, 40, 19, 20]. Thus, from results proved in [20], it follows that in this extension of Bylander's formulation, PLAN LENGTH is NP-complete.

5.2.3 Datalog Operators

Below, we no longer restrict the predicates to be propositions. As a result, planning is much more complex than in the previous case.

In datalog planning, the number of ground terms we have is pc^a, where p is the number of predicates, c is the number of constants, and a is the arity of predicates. This value is exponential in terms of the size of the input. Each state is a subset of ground terms, and hence the number of states is double exponential. In the unrestricted case, we need to search through this space, requiring a doubly exponential amount of work. Since the size of a state is at most exponential, we can make a nondeterministic forward search starting with the initial state, and solve the problem in EXPSPACE.[11] This is stated formally below.

[11] Although it has not been proved, it is believed that EXPSPACE equals double exponential time.

Theorem 5.7 PLAN EXISTENCE *is* EXPSPACE*-complete. It is still* EXPSPACE*-complete if we restrict* **P** *to be positive.*

When we restrict **P** to be deletion-free, we still need to search through the same space as before. However, now we have a monotonicity property. Since all delete-lists are empty, what ever is asserted at a step in a plan remains true after that point. Hence no operator instance needs to appear in a plan more than once, as the latter appearances would not have any affect. The number of operator instances is exponential, and all we need to do is to non-deterministically guess a sequence of operator instances, and verify it. Thus the problem is NEXPTIME-complete, as stated below.

Theorem 5.8 *If we restrict* **P** *to be deletion-free, then* PLAN EXISTENCE *is* NEXPTIME*-complete.*

When we restrict **P** to be both positive and deletion-free, then just as above, each operator instance needs to appear in a plan at most once. In addition, the ordering of the operators in a plan does not matter as long as their preconditions are satisfied. The reason for this is as follows: since **P** is deletion-free, whatever is asserted remains asserted; and since **P** is positive, all the operators have only positive preconditions; and thus any operator that is executable at some point in the plan remains executable at subsequent points in the plan. As a result, we can keep executing operator instances, until we reach the goal, or all the executable operator instances have been used. Since this takes exponential time, we have the following result.

Theorem 5.9 *If we restrict* **P** *to be positive and deletion-free, then* PLAN EXISTENCE *is* EXPTIME*-complete.*

Now, in addition to the above restrictions, suppose we require each planning operator to have at most one precondition, which must be positive. Then we can do backward chaining, starting with the set of goals, and at each step on-deterministically choosing an operator instance, removing the subgoals it adds, and inserting the precondition of the operator as a new subgoal. Each new operator achieves at least one subgoal and introduces at most one new subgoal, so the size of the set of unachieved goals is monotonically non-increasing. Furthermore, since **P** is positive and deletion-free, no operator will clobber a previously achieved subgoal, so we do not need to keep track of subgoals that have already been achieved. Thus, we can solve PLAN EXISTENCE in PSPACE. More formally, we have the following result.

Theorem 5.10 *If we restrict* **P** *to be context-free, positive, and deletion-free, then* PLAN EXISTENCE *is* PSPACE*-complete.*

We now examine the complexity of PLAN LENGTH.

Theorem 5.11 *If we restrict* **P** *to be deletion-free, positive, and context-free, then* PLAN LENGTH *is* PSPACE*-complete.*

Here is a brief explanation of the above result. Since **P** is deletion-free, no operator need to appear more than once in a plan. Thus, we can show that PLAN EXISTENCE is a special case of PLAN LENGTH, with $k =$ the number of operator instances. PLAN EXISTENCE was proved to be PSPACE-hard (Theorem 5.10), hence hardness follows. For proving membership, remember that the algorithm we provided for the existence problem was nondeterministic. Whenever this is the case, we can always introduce a counter to keep track of number of operators in the plan, and fail when it exceeds k.

Theorem 5.12 PLAN LENGTH *is* NEXPTIME*-complete in each of the following cases:*

1. **P** *is deletion-free and positive;*
2. **P** *is deletion-free;*
3. **P** *is positive;*
4. *no restrictions (except, of course, that* **P** *is function-free).*

The reason for the above result is as follows. For membership, notice that the length of the plan is bound by k, which is part of the input. Since k is encoded in binary, it will confine us to plans of at most exponential length. Thus we can solve the problem in NEXPTIME. For the hardness result, we only need to discuss case 1, which is a special case of the other cases. Remember that PLAN EXISTENCE is EXPTIME-complete in this case, because of the property that the ordering of operators does not matter as long as all the preconditions are satisfied. This property allowed us to do a forward search, arbitrarily choosing the next operator. However, in the case of PLAN LENGTH, we can not choose the operators arbitrarily: we need to choose them so that the plan length does not exceed k. This makes the problem harder.

When we have an overall look at the results in this section, we note that if delete lists are allowed, then PLAN EXISTENCE is EXPSPACE-complete but PLAN LENGTH is only NEXPTIME-complete. Normally, one would not expect PLAN LENGTH to be easier than PLAN EXISTENCE, and if we look at Table 2, this is true in all cases except this one. The reason for this anomaly is that the length of a plan can sometimes be doubly exponential in the length of the input. In PLAN LENGTH we are given a bound k, encoded in binary, which confines us to plans of length at most exponential in terms of the input. Hence in the worst case of PLAN LENGTH, finding the plan is easier than in the worst case of PLAN EXISTENCE.

We do not observe the same anomaly in the propositional cases described in Section 5.2.1, because in those cases the lengths of the plans are at most exponential in the length of the input, so giving an exponential bound on the length of the plan does not reduce the complexity of PLAN LENGTH. As a result, in the propositional case, both PLAN EXISTENCE and PLAN LENGTH are PSPACE-complete.

5.3 Planning When the Operator Set is Fixed

The results in Section 5.2 were for the case in which the set of operators is part of the input. However, in many well known planning problems, the set of operators is fixed in advance. For example, in the blocks world (see Example 5.1), we have only four operators: stack, unstack, pickup and putdown.

In this section we will present complexity results on planning problems in which the set of operators is fixed, and only the initial state and goal are allowed to vary. The problems we will consider will be of the form: "given the initial state S_0 and the goal G, is there a plan that achieves G?" We assume every predicate symbol appearing in G and S_0 appears in at least one of the planning operators. This restriction is reasonable because the operators can neither add nor delete atoms constructed from any other predicate symbols.

5.3.1 Propositional Operators

Propositional planning with a fixed set of operators is very restrictive. The number of possible plans is constant. We include the following two results just for the sake of completeness.

Theorem 5.13 PLAN EXISTENCE *can be solved in constant time if we restrict* $\mathbf{P} = (S_0, \mathcal{O}, G)$ *to be propositional and* $\mathcal{O}$ *to be a fixed set.*

Corollary 5.5 PLAN LENGTH *can be solved in constant time if we restrict* $\mathbf{P} = (S_0, \mathcal{O}, G)$ *to be propositional and* $\mathcal{O}$ *to be a fixed set.*

5.3.2 Datalog Operators

The number of ground instances of predicates is pc^a, where p is the number of predicates, c is the number of constants, and a is the arity of the predicates. When the set of operators is fixed, a will be a constant value, hence we will have a polynomial number of ground instances of predicates. These can be mapped into propositions in polynomial time, providing a reduction from datalog planning with a fixed set of operators to propositional planning with a varying set of operators. Note that this reduction will also preserve the length of the plans. Thus datalog planning with a fixed set of operators has the same complexity as propositional planning with varying sets of operators, as stated in the following theorem.

Theorem 5.14

1. *If we restrict* $\mathbf{P}$ *to be fixed, deletion-free, context-free and positive, then* PLAN EXISTENCE *is in* NLOGSPACE *and* PLAN LENGTH *is in* NP.
2. *If we restrict* $\mathbf{P}$ *to be fixed, deletion-free, and positive, then* PLAN EXISTENCE *is in* P *and* PLAN LENGTH *is in* NP.
3. *If we restrict* $\mathbf{P}$ *to be fixed and deletion-free, then* PLAN EXISTENCE *and* PLAN LENGTH *are in* NP.
4. *If we restrict* $\mathbf{P}$ *to be fixed, then* PLAN EXISTENCE *and* PLAN LENGTH *are in* PSPACE.

The above theorem puts a bound on how hard planning can be with a fixed set of operators. Naturally, the exact complexity of the problem depends on which particular fixed set of operators we are dealing with. The following theorems state that we can find fixed sets of operators such that their corresponding planning problems are complete for the complexity classes mentioned in the previous theorem.

Theorem 5.15 *There exists a fixed positive deletion-free set of operators* $\mathcal{O}$ *for which* PLAN LENGTH *is* NP*-hard.*

Theorem 5.16 *There exist fixed deletion-free sets of operators* $\mathcal{O}$ *for which* PLAN EXISTENCE *and* PLAN LENGTH *are* NP*-hard.*

Theorem 5.17 *There exists a fixed set of positive operators* $\mathcal{O}$ *for which* PLAN EXISTENCE *and* PLAN LENGTH *are* PSPACE*-hard.*

Note that all three of the previous theorems prove hardness for *some* fixed sets of operators. For some other sets of operators, the problem might be much easier, even constant time. (e.g. think of an empty set of operators)

5.3.3 Conditional Operators

In Proposition 3.4, we showed that if the planning operators are extended to allow conditional effects, this does not affect our decidability and undecidability results. The following theorem makes the same statement about our complexity results. As with Proposition 3.4, this theorem is stated in a rather unconventional way, in order to avoid duplicating the statements of the fifteen theorems mentioned in it.

Theorem 5.18 *(Complexity of planning with conditional operators) Theorems 5.3 through 5.5, 5.7 through 5.17, and their corollaries still hold when $\mathcal{O}$ is allowed to contain conditional operators.*

The fact that conditional operators do not affect the complexity should not be surprising. In a single-agent static world with complete information, one does not need conditional actions. Conditional operators are useful only when we have incomplete information about the initial state of the world, or the affects of the operators, so that we can try to come up with a plan that would work in any situation that is consistent with the information available. Otherwise, we can replace the conditional operators with a number of ordinary STRIPS-style operators, as described in Section 3.4.2, to obtain an equivalent planning domain. Although this reduction is sufficient for proving that conditional operators do not affect our decidability and undecidability results (Proposition 3.4), it is not sufficient to prove Theorem 5.18, because there are an exponential number of combinations, and thus the reduction is not polynomial. However, with minor modifications, the proofs of Theorems 5.2 through 5.17 will do the job.

6 Related Work

6.1 Planning

Bylander has done several studies on the complexity of propositional planning [5, 6]. We have stated some of his results in Theorem 5.2 and Table 2. More recently, he has studied the complexity of propositional planning extended to allow a limited amount of inference in the domain theory [6]. His complexity results for this case range from polyomial time to PSPACE-complete.

Chapman was the first to study issues relating to the undecidability of planning; we have discussed his work in detail in Section 4.

Backstrom and Klein found a class of planning problems called SAS-PUBS, for which planning can be done in polynomial time [2]. Their planning formalism is somewhat different from ours: they make use of *state variables* that take values from a finite set, and consider a planning state to be an assignment of values to these state variables. Since they restrict each state variable to have a domain of exactly two values, we can consider each state variable to be a proposition; thus, in effect they are doing propositional planning. However, their operators have further restrictions: they restrict each operator to change at most one state variable, and do not allow more than one operator to change a state variable to a given value. Their restrictions are so strict that they were unable to find any domains (not even blocks world) that they could represent in their formalization. They tried to overcome this problem by weakening some of their restrictions, making the complexity of their algorithm go to exponential time—but still could not find any reasonable domain. It is not very easy to compare our results with theirs, because we use a different formalism—but we can safely state that we analyze a much broader range of problems, and we require less severe restrictions to get polynomial-time results.

Korf [22] has pointed out that given certain assumptions, one can reduce exponentially the time required to solve a conjoined-goal planning problem, provided that the individual goals are independent. Yang, Nau, and Hendler [42] have generalized this result by showing that one can still exponentially reduce the time required for planning even if the goals are not independent, provided that only certain kinds of goal interactions are allowed. Under this same set of goal interactions, they have also developed some efficient algorithms for merging plans to achieve multiple goals [41, 42].

Complexity results have been developed for blocks-world planning by Gupta and Nau [19, 20] and also by Chenoweth [9]. Gupta and Nau [19, 20] have shown that the complexity of blocks-world planning arises not from deleted-condition interactions as was previously thought, but instead from enabling-condition interactions. Their speculations that enabling-condition interactions are important for planning in general seem to be corroborated by some of our results, as discussed in Section 7.2 below.

6.2 Temporal Projection

Another problem that is closely related to planning is the problem of temporal projection, or what Chapman calls the "modal truth" of an atom [8]. Given an atom a, an initial state S_0, and a partially ordered set of actions P, the question is whether a is necessarily/possibly true after execution of P. This question is especially important in partial-order planners such as NOAH [13], NONLIN [14], and SIPE [15]. For example, McDermott [25] says "unfortunately, partial orders have a big problem, that there is no way of deciding what is true for sure before a step without considering all possible step sequences consistent with the current partial order," and Pednault [30] also expresses similar sentiments.

One problem is what it means for a to be necessarily true if some of the total orderings of P are unexecutable. Chapman [8] assumes that a is necessarily true after executing P only if every total ordering of P is both executable and achieves a; and in return, he comes up with a polynomial-time algorithm for determining the necessary truth of a. However, his algorithm does not work correctly for establishing the possible truth of a (in a paper currently in progress, we prove that problem is NP-hard).

Chapman also proves that with conditional planning operators, establishing the necessary truth of a is co-NP-hard; and Dean and Boddy [10] prove a similar result with a more general notion of conditional planning operators (the same definition we use in Section 3.4.2).[12] Dean and Boddy [10] also try to come up with approximate solutions for the problem. They present algorithms for computing a subset of the propositions that are necessarily true, and for computing a superset of the propositions that are possibly true. Furthermore, the complexity of these algorithms is polynomial if the number of triples for each operator is bounded with a constant. However, we do not know of any results concerning how close the approximations are.

7 Conclusion

In this paper, we have studied the decidability and complexity of planning with STRIPS-style planning operators (i.e., operators comprised of preconditions, add lists, and delete

[12] In both cases, they state that the problem is NP-hard, but their proofs establish co-NP-hardness instead.

lists). Our results show that planning is a hard problem even under severe restrictions on the nature of planning domains. We have been able to classify sets of problems in terms of syntactic domain parameters, establish the decidability and computational complexity of each of these classes, and gain insight into why and how these classes of problems are so hard.

7.1 Decidability and Undecidability

We have proved equivalence theorems relating definite logic programs to planning with positive, deletion-free operators. This equivalence allows us to transport many results from logic programming to planning, leading to a number of decidability and undecidability results, as summarized in Table 1. If we use the conventional definitions of a first-order language and a state (i.e., the language contains only finitely many constant symbols and all states are finite), then whether or not PLAN EXISTENCE is decidable depends largely on whether or not function symbols are allowed:

- If the language is allowed to contain function symbols (and hence infinitely many ground terms), then, in general, PLAN EXISTENCE is undecidable, regardless of whether or not the planning domain is positive, deletion-free, and context-free. However, if the planning domains are restricted to be weakly recurrent, and only bounded goals are considered, then PLAN EXISTENCE is decidable even in the presence of function symbols.

- If the language does not contain function symbols (and hence has only finitely many ground terms), then PLAN EXISTENCE is decidable, regardless of whether or not the planning domain is positive, deletion-free, and context-free.

For comparison with Chapman's [8] results, Table 1 also includes decidability and undecidability results for the cases where we allow infinitely many constant symbols, infinite initial states, and operators with conditional effects. These results relate to Chapman's work as follows:

1. They solve an open problem posed in [8], regarding the decidability of planning if infinitely many constants are allowed. Unless **P** is restricted to be positive or deletion-free, the problem is undecidable.

2. It clarifies one of the results in [8]. In particular, whether or not the definition of a planning domain is extended to allow conditional planning operators makes no difference in any of our decidability and undecidability results.

When certain syntactic (predicate and atomic acyclicity) and semantic properties (weak-recurrence) are satisfied by positive, deletion-free planning domains (even those containing function symbols), we have proved, in addition, that plan existence for bounded goals is decidable.

7.2 Complexity

Based on various syntactic criteria on what planning operators are allowed to look like, we have developed a comprehensive theory of the complexity of planning; the results are summarized in Table 2. Examination of this table reveals several interesting properties:

1. Comparing the complexity of PLAN EXISTENCE in the propositional case (in which all predicates are restricted to be 0-ary) with the datalog case (in which the predicates may have constants or variables as arguments) reveals a regular pattern. In most cases, the complexity in the datalog case is exactly one level harder than the complexity in the corresponding propositional case. We have EXPSPACE-complete versus PSPACE-complete, NEXPTIME-complete versus NP-complete, EXPTIME-complete versus polynomial.

2. If delete lists are allowed, then PLAN EXISTENCE is EXPSPACE-complete but PLAN LENGTH is only NEXPTIME-complete. Normally, one would not expect PLAN LENGTH to be easier than PLAN EXISTENCE. In this case, it happens because the length of a plan can sometimes be doubly exponential in the length of the input. In PLAN LENGTH we are given a bound k, encoded in binary, which confines us to plans of length at most exponential in terms of the input. Hence in the worst case of PLAN LENGTH, finding the plan is easier than in the worst case of PLAN EXISTENCE.

   We do not observe the same anomaly in the propositional case, because the lengths of the plans are at most exponential in the length of the input. Hence, giving an exponential bound on the length of the plan does not reduce the complexity of PLAN LENGTH. As a result, in the propositional case, both PLAN EXISTENCE and PLAN LENGTH are PSPACE-complete.

3. When the operator set is fixed in advance, any operator whose predicates are not all propositions can be mapped into a set of operators whose predicates are all propositions. Thus, planning with a fixed set of datalog operators has basically the same complexity as planning with propositional operators that are given as part of the input.

4. PLAN LENGTH has the same complexity regardless of whether or not negated preconditions are allowed. This is because what makes the problem hard is how to handle *enabling-condition interactions*, i.e., how to choose operators that achieve several subgoals in order to minimize the overall length of the plan [20], and this task remains equally hard regardless of whether negated preconditions are allowed.

5. Delete lists are more powerful than negated preconditions. Thus, if the operators are allowed to have delete lists, then whether or not they have negated preconditions has no effect on the complexity.

Below, we summarize how and why our parameters affect the complexity of planning:

- If no restrictions are put on **P**, any operator instance might need to appear many times in the same plan, forcing us to search through all the states, which are double exponential in number. Since the size of any state is at most exponential, PLAN EXISTENCE can be solved in EXPSPACE.

- When **P** is restricted to be deletion-free, any predicate instance asserted remains true throughout the plan, hence no operator instance needs to appear in the same plan twice. Since the number of operator instances is exponential, this reduces the complexity of PLAN EXISTENCE to NEXPTIME.

- When **P** is further restricted to be positive, we get the nice property that no operator clobbers another. Thus the order of the operators in the plan does not matter, and the complexity of PLAN EXISTENCE reduces to EXPTIME.

- Inspite of the restrictions above, PLAN LENGTH remains NEXPTIME. Since we try to find a plan of length at most k, which operator instances we choose, and how we order them makes a difference.

- When $\mathbf{P}$ is also restricted to be context-free, we can do backward search, and since each operator has at most one precondition, the number of the subgoals does not increase. Thus both PLAN EXISTENCE and PLAN LENGTH with these restrictions can be solved in PSPACE.

- The previous arguments also hold for propositional planning, with the exception of the anomaly in the unrestricted case for PLAN LENGTH, which we have discussed before. As a result of restricting predicates to be 0-ary, the number of operator instances, the size of states reduce to polynomial from exponential, hence in general, the complexity results for propositional planning are one level lower than the complexity results with datalog operators.

7.3 Future Work

Although our equivalence between planning and logic programming only holds in certain limited cases, this equivalence has allowed us to transport many results from logic programming to planning. It is not a trivial task to extend this equivalence, because negation has different semantics for logic programming and planning—but it is certainly worth investigating, and we intend to do so in the future.

Although much research has been done on more general operator representations such as those used in hierarchical nonlinear planning, most theoretical studies of planning have been confined to planning with strips-like operators. As a result, much of the current work in planning is without much theoretical basis. For example, in his paper on regression planning [25], McDermott states that

> ... there are two main choices in the space of refinement planners: (1) a heuristic, nonlinear, progressive planner, and (2) a rigorous, linear, regressive planner.

In the conclusion of his paper, McDermott continues:

> For the time being, practical work on planning will continue to focus on nonlinear planning, because all planning algorithms are exponential.
>
> ...But theoretical work in the field should go on, if for no other reason than that it might inspire us to come up with a theory of nonlinear planning in realistic domains, which is so far entirely lacking. ...

The next task we intend to undertake is to develop a formalization of hierarchical nonlinear planning, and to investigate how difficult hierarchical nonlinear planning is, and how to develop more efficient algorithms.

Acknowledgement

We appreciate the useful comments about this paper that we received from Tom Bylander and from the referees.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1976.

[2] Christer Backstrom, and Inger Klein. "Planning in polynomial time: the SAS-PUBS class" *Computational Intelligence* **7** (1991), pp. 181–197.

[3] M. Bezem. "Characterizing Termination of Logic Programs with Level Mappings." In E. Lusk and R. Overbeek, editors, *Proc. 1989 North American Conf. on Logic Programming*, pp. 69–80, MIT Press.

[4] H.A. Blair. "Canonical Conservative Extensions of Logic Program Completions," *Proc. 4th IEEE Symposium on Logic Programming*, 1989, pp. 154–161.

[5] T. Bylander. "Complexity Results for Planning," *Proc. IJCAI-91*, 1991, pp. 274–279.

[6] T. Bylander. "Complexity Results for Extended Planning," In *Proc. First Internat. Conf. AI Planning Systems*, 1992,

[7] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, MA, 1985.

[8] D. Chapman. "Planning for Conjunctive Goals," *Artificial Intelligence* **32**, 1987, pp. 333-377.

[9] Stephen V. Chenoweth. On the NP-hardness of blocks world. In *AAAI-91: Proc. Ninth National Conf. Artificial Intelligence*, pp. 623–628, July 1991.

[10] Thomas Dean, and Mark Boddy. "Reasoning about Partially Ordered Events" Artificial Intelligence Journal, 1988 pp. 375–399.

[11] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. When is planning decidable? In *Proc. First Internat. Conf. AI Planning Systems*, 1992, pp. 222–227.

[12] Richard E. Fikes, Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 88–97. Morgan Kaufman, 1990.

[13] Earl D. Sacerdoti. "The Nonlinear Nature of Plans." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 162–170. Morgan Kaufman, 1990.

[14] Austin Tate. "Generating Project Networks" In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 291–296. Morgan Kaufman, 1990.

[15] David E. Wilkins. "Domain independent Planning: Representation and Plan Generation" In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 319–335. Morgan Kaufman, 1990.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[17] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: a Foundation for Computer Science*. Addison-Wesley, 1989.

[18] C. Green. "Application of Theorem-Proving to Problem Solving," *Proc. IJCAI-69*, 1969.

[19] Naresh Gupta and Dana S. Nau. "Complexity Results for Blocks-World Planning," *Proc. AAAI-91*, 1991. Honorable mention for the best paper award.

[20] Naresh Gupta and Dana S. Nau, "On the Complexity of Blocks-World Planning," *Artificial Intelligence* **56**:2–3 (1992), pp. 223–254.

[21] Kluzniak and Szapowicz. extract from APIC studies in data processing no. 24. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 140–153. Morgan Kaufman, 1990.

[22] Korf, R.E., "Planning as Search: A Quantitative Approach," *Artificial Intelligence* **33**, 1987, 65-88.

[23] Vladimir Lifschitz. "On the Semantics of STRIPS." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 523–530. Morgan Kaufman, 1990.

[24] J. W. Lloyd. *Foundations of Logic Programming*, Springer Verlag, 1987.

[25] Drew McDermott. "Regression Planning" *International Journal of Intelligent Systems* **6** (1991), pp. 357–416.

[26] S. Minton, J. Bresna and M. Drummond. "Commitment strategies in planning," *Proc. IJCAI-91*, 1991.

[27] D. McAllester and D. Rosenblitt. "Systematic nonlinear planning," *Proc. AAAI-91*, July 1991.

[28] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, 1980.

[29] Earl D. Sacerdoti. "Planning in a hierarchy of abstraction spaces." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 98–108. Morgan Kaufman, 1990. Originally appeared in *Artificial Intelligence* **5** (1974), 115–135.

[30] Edwin P. D. Pednault. "Synthesizing Plans that Contain Actions with Context-dependent Effects" *Computational Intelligence* **4** (1988), pp. 356–372.

[31] M. A. Peot. "Conditional Nonlinear Planning" In *Proc. First Internat. Conf. AI Planning Systems*, 1992, pp. 189–197.

[32] D.A. Plaisted. "Complete Problems in the First-Order Predicate Calculus," *Jour. Computer and Systems Sciences* **29** (1984), pp. 8–35.

[33] Earl D. Sacerdoti. "The nonlinear nature of plans." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 206–214. Morgan Kaufman, 1990. Originally appeared in *Proc. IJCAI-75.*

[34] J. Sebelik and P. Stepanek. "Horn Clause Programs for Recursive Functions." In K. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pp. 325–340. Academic Press, 1980.

[35] J. Shoenfield. *Mathematical Logic*, Academic Press, 1967.

[36] L. J. Stockmeyer and A. K. Chandra. "Provably Difficult Combinatorial Games," RC 6957, IBM T. J. Watson Research Ctr., 1978.

[37] G.J. Sussman. *A Computational Model of Skill Acquisition.* American Elsevier, New York, 1975.

[38] A. Tate, J. Hendler, and M. Drummond. "A review of AI planning techniques." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 26–49. Morgan Kaufman, 1990.

[39] M. Vardi. "The Complexity of Relational Query Languages," *Proc. 14th ACM Symp. on Theory of Computing*, San Francisco, 1982, pp. 137–146.

[40] R. Waldinger. "Achieving several goals simultaneously." In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pp. 118–139. Morgan Kaufman, 1990. Originally appeared in *Machine Intelligence 8.*

[41] Q. Yang, D. S. Nau, and J. Hendler. Optimization of multiple-goal plans with limited interaction. In *Proc. DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, 1990.

[42] Q. Yang, D. S. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, to appear.

A Proofs of Decidability and Undecidability Results

A.1 Equivalence between Logic Programming and Planning

Lemma A.1 *Suppose that* $\mathbf{P} = (S_0, \mathcal{O})$ *is any* positive, deletion-free *planning domain, and*

$$S_0 \overset{\alpha_1,\theta_1}{\Longrightarrow} S_1 \overset{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \overset{\alpha_n,\theta_n}{\Longrightarrow} S_n$$

is a plan that achieves some goal G (we really don't care what G is as far as this lemma is concerned). Then:

1. $S_0 \subseteq S_1 \subseteq S_2 \cdots \subseteq S_n$.
2. *If operator α is θ-executable in state S_j, then α is θ-executable in state S_k for all $k \geq j$.*

Proof.

1. Immediate consequence of the fact that $\mathrm{Del}(\alpha) = \emptyset$ for all $\alpha \in \mathcal{O}$. Hence, for all $0 \leq i \leq n-1$,
$$S_{i+1} = S_i \cup \mathrm{Add}(\alpha_i)\theta_i.$$
2. Suppose α is θ-executable in state S_j. Then $\mathrm{Pre}(\alpha)\theta \subseteq S_j \subseteq S_k$. As $\mathrm{Pre}(\alpha)$ is negation-free, the condition that $\{B\theta : \neg B \text{ is a negated atom in } \mathrm{Pre}(\alpha)\} \cap S_k = \emptyset$ is immediately satisfied and hence α is executable in state S_k.

■

Before proving the following theorem, we need to introduce an operator, called T_P, associated with any logic program P. The operator, which is well-known in logic programming, maps states (i.e. Herbrand interpretations) to states. Intuitively, given an interpretation I, $T_P(I)$ is the (smallest) interpretation obtained as follows: if there is a clause in P having a ground instance C with A in the head, and whose body is satisfied by interpretation I, then $A \in T_P(I)$. In particular, I may not necessarily be a subset of $T_P(I)$[24].

Definition A.1 Given a logic program P that contains no negative atoms in the body of any clause, T_P is an operator associated with P that maps sets of ground atoms to sets of ground atoms as follows: $T_P(I) = \{A : A$ is a ground atom and there is a clause in P having a ground instance of the form $A \leftarrow B_1 \,\&\ldots\&\, B_n$ such that $\{B_1, \ldots, B_n\} \subseteq I\}$.

When we start deducing ground atoms from a program, nothing is initially known to be true. Applying the T_P operator once, we know that all ground instances of *facts* are true. Repeating this process one step further, we can conclude all the facts as well as all the ground atoms that can be deduced by applying one rule. The following definition specifies how T_P may be iterated *upwards* in this way starting from the empty set of atoms:

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow (n+1) &= T_P(T_P \uparrow n) \\ T_P \uparrow \omega &= \bigcup_{n<\omega} T_P \uparrow n \end{aligned}$$

It turns out that $T_P \uparrow \omega$ is the least fixed point of T_P for those programs P that contain no negations in clause bodies.

Theorem 3.1 (Equivalence Theorem I) Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a positive, deletion-free planning domain and G is a goal. Then there is a plan to achieve G from $\mathbf{P}$ iff $\mathrm{LP}(\mathbf{P}) \models G$.

Proof. Let $G = (\exists)(A_1 \,\&\ldots\&\, A_s)$.

$(\Rightarrow)$: Suppose $\mathrm{LP}(\mathbf{P}) \models G$. Then there is a ground instance, $G\sigma$ of G such that $\mathrm{LP}(\mathbf{P}) \models G\sigma$ and an integer $n < \omega$ such that $T_{\mathrm{LP}(\mathbf{P})} \uparrow n \models G\sigma$, i.e. $\{A_1\sigma, \ldots, A_s\sigma\} \subseteq T_{\mathrm{LP}(\mathbf{P})} \uparrow n$. We proceed by induction on n.

Base Case ($n = 1$). In this case, for each $1 \leq i \leq s$, there is a clause in $\mathrm{LP}(\mathbf{P})$ having a ground instance of the form

$$A_i\sigma \leftarrow .$$

Consider an arbitrary A_i, $1 \leq i \leq s$. The unit clause

$$A_i\sigma \leftarrow .$$

could have been placed in $\mathrm{grd}(\mathrm{LP}(\mathbf{P}))$ for one of two reasons:
Case 1: $A_i\sigma$ is in S_0 or
Case 2: There is a planning operator α such that $\mathrm{Pre}(\alpha) = \emptyset$ and $\mathrm{Add}(\alpha)$ contains an atom A'_i such that $A_i\sigma$ is a ground instance of A'_i (via an mgu σ_i, say).

Thus, the set $X = \{A_1\sigma, \ldots, A_s\sigma\}$ can be partitioned into two parts: The set X_1 consisting of those atoms satisfying case 1 above, and the set X_2 of those atoms that do not satisfy Case 1 above (and hence must satisfy case 2 above).

Suppose $X_2 = \{A_{\rho(1)}\sigma, \ldots, A_{\rho(r)}\sigma\}$ where $0 \leq r \leq s$. Then

$$S_0 \overset{\alpha_{\rho(1)},\sigma_{\rho(1)}}{\Longrightarrow} S_1 \cdots S_{r-1} \overset{\alpha_{\rho(r)},\sigma_{\rho(r)}}{\Longrightarrow} S_r$$

is a planning sequence such that $A_1\sigma \,\&\ldots\&\, A_s\sigma$ is true in S_r. To see this, observe that every $A_i\sigma \in X_1$ is true in S_0 and hence must be true in S_r by Lemma A.1. Likewise, every $A_{\rho(j)} \in X_2$ is true in $S_{\rho(j)} \subseteq S_r$ by Lemma A.1.

Inductive Case $(n+1)$. Suppose $T_{\mathrm{LP}(\mathbf{P})} \uparrow (n+1) \models G\sigma$. Then for each $1 \leq i \leq s$, there is a clause $C_i \in \mathrm{grd}(\mathrm{LP}(\mathbf{P}))$ of the form

$$A_i\sigma \leftarrow B_1^i \,\&\ldots\&B_{h_i}^i$$

such that $B_1^i \,\&\ldots\&B_{h_i}^i$ is true in $T_{\mathrm{LP}(\mathbf{P})} \uparrow n$. By the induction hypothesis, for all $1 \leq i \leq s$, there is a planning sequence, $\Re_i$ that achieves the goal $(B_1^i\&\ldots\&B_{h_i}^i)$. Clause C_i is obtained from a planning operator α_i by applying a ground substitution θ_i to a clause in $\mathrm{LP}(\alpha)$. Hence,

$$\Re_i \overset{\alpha_i,\theta_i}{\Longrightarrow} \mathbf{S}_i$$

would be a planning sequence that achieves $A_i\sigma$ (where $\mathbf{S}_i$ is the state that results by θ_i-executing α_i in the last state of $\Re_i$). Call the above planning sequence $\wp_i$.

Clearly, each $\wp_i$ achieves goal $A_i\sigma$. The only remaining problem is to put together the planning sequence $\wp_i$ in such a way that we achieve the conjunctive goal $(A_1\sigma \,\&\ldots\&\, A_s\sigma)$. We do this as follows.

We first show how to put $\wp_1$ and $\wp_2$ together to get a plan that achieves $(A_1\sigma \,\&\, A_2\sigma)$. Suppose $\wp_1$ is the sequence

$$S_0 \overset{\alpha_{v(1)},\theta_{v(1)}}{\Longrightarrow} S_1^1 \overset{\alpha_{v(2)},\theta_{v(2)}}{\Longrightarrow} \cdots \overset{\alpha_{v(k_1)},\theta_{v(k_1)}}{\Longrightarrow} S_{k_1}^1$$

and $\wp_2$ is the sequence

$$S_0 \overset{\alpha_{w(1)},\theta_{w(1)}}{\Longrightarrow} S_1^2 \overset{\alpha_{w(2)},\theta_{w(2)}}{\Longrightarrow} \cdots \overset{\alpha_{w(k_2)},\theta_{w(k_w)}}{\Longrightarrow} S_{k_2}^2.$$

The following is a valid planning sequence that achieves $(A_1\sigma \,\&\, A_2\sigma)$.

$$\begin{array}{lll}
S_0 & \overset{\alpha_{v(1)},\theta_{v(1)}}{\Longrightarrow} & S_1^1 \\
 & \overset{\alpha_{w(1)},\theta_{w(1)}}{\Longrightarrow} & S_1^1 \cup S_1^2 \\
 & \overset{\alpha_{v(2)},\theta_{v(2)}}{\Longrightarrow} & S_1^1 \cup S_1^2 \cup S_2^1 \\
 & \overset{\alpha_{w(2)},\theta_{w(2)}}{\Longrightarrow} & S_1^1 \cup S_1^2 \cup S_2^1 \cup S_2^2 \\
 & \cdots & \\
 & \overset{\alpha_{v(k_1)},\theta_{v(k_1)}}{\Longrightarrow} & \left(\bigcup_{j=1}^{k_1} S_j^1\right) \cup \left(\bigcup_{z=1}^{k_2-1} S_z^2\right) \\
 & \overset{\alpha_{w(k_2)},\theta_{w(k_2)}}{\Longrightarrow} & \left(\bigcup_{j=1}^{k_1} S_j^1\right) \cup \left(\bigcup_{z=1}^{k_2} S_z^2\right).
\end{array}$$

The above sequence achieves the goal $(A_1\sigma \,\&\, A_2\sigma)$. To see this, observe the following:

1. Each of the $\alpha_{v(i)}$'s is $\theta_{v(i)}$-executable in the state S^1_{i-1} and hence, by Lemma A.1, it is also $\theta_{v(i)}$-executable in the state $\bigcup_{j=1}^{i} S^1_j \;\cup\; \bigcup_{h=1}^{i-1} S^2_h$. The same reasoning applies to the $\alpha_{w(j)}$'s.

   To see how the states generated in the above planning sequence are constructed, we now explain how we construct the state $(S^1_1 \cup S^2_1)$ after $\theta_{w(1)}$-execution of α_{w_1} in state S^1_1. In the above sequence, $\theta_{v(1)}$-execution of $\alpha_{v(1)}$ in state S_0 leads to state S^1_1 (according to $\wp_1$). Furthermore, as $\alpha_{v(1)}$ is positive and deletion-free, we know that $S_0 \subseteq S^1_1$. As all operators in $\mathbf{P}$ are positive and deletion-free, any operator that is θ-executable in state S_0 is also θ-executable in any state S' such that $S \subseteq S'$. As α_{w_1} is $\theta_{w(1)}$-executable in state S_0 and as $S_0 \subseteq S^1_1$, we know that α_{w_1} is $\theta_{w(1)}$-executable in state S^1_1 as well, resulting in state $(S^1_1 \cup \mathit{Add}(\alpha_{w_1}))$. But from $\wp_2$, we know that $\mathit{Add}(\alpha_{w_1}) \subseteq (S^2_1 - S_0)$. Hence, $(S^1_1 \cup \mathit{Add}(\alpha_{w_1})) \subseteq (S^1_1 \cup (S^2_1 - S_0))$. As $S_0 \subseteq S^1_1$, $(S^1_1 \cup \mathit{Add}(\alpha_{w_1})) \subseteq (S^1_1 \cup S^2_1)$.

   It can similarly be shown that $(S^1_1 \cup S^2_1) \subseteq (S^1_1 \cup \mathit{Add}(\alpha_{w_1}))$. Suppose not. Then there must be an $A \in S^2_1$ such that $A \notin (S^1_1 \cup \mathit{Add}(\alpha_{w_1}))$. As $A \in S^2_1 = S_0 \cup \mathit{Add}(\alpha_{w_1})$, there are two cases. If $A \in S_0$, then $A \in S^1_1$ because $S_0 \subseteq S^1_1$. This contradicts out assumption that $A \notin (S^1_1 \cup \mathit{Add}(\alpha_{w_1}))$. If $A \in \mathit{Add}(\alpha_{w_1})$, then we likewise obtain a contradiction. Hence, the state obtained by $\theta_{w(1)}$-execution of α_{w_1} in state S^1_1 is $(S^1_1 \cup S^2_1)$.

2. As $A_1\sigma \in S^1_{k_1}$ and as $A_2\sigma \in S^2_{k_2}$, it follows that $\{A_1\sigma, A_2\sigma\} \subseteq \left(\bigcup_{j=1}^{k_1} S^1_j\right) \cup \left(\bigcup_{z=1}^{k_2} S^2_z\right)$.

To achieve the goal $(A_1\sigma \,\&\, A_2\sigma \,\&\, A_3\sigma)$, we simply repeat the same process by combining together the above sequence with $\wp_3$. On iterating this process till we have finished processing $\wp_s$, we would have a plan that achieves $(A_1\sigma \,\&\ldots\&\, A_s\sigma)$.

$(\Leftarrow)$: Suppose on the other hand, that there is a plan $\wp$ that achieves $G = (\exists)(A_1\&\ldots\&A_s)$ from $\mathbf{P}$. The $\wp$ must be of the form:

$$S_0 \stackrel{\alpha_1,\sigma_1}{\Longrightarrow} S_1 \cdots \stackrel{\alpha_{r-1},\sigma_{r-1}}{\Longrightarrow} S_r.$$

We proceed by induction on r.

Base Case ($r = 0$). In this case, G is true in S_0 itself. As each clause in S_0 is in LP($\mathbf{P}$), G is true in $T_P \uparrow 1$ and hence is entailed by P.

Inductive Case ($r = t + 1$). Suppose our plan is of the form

$$S_0 \stackrel{\alpha_1,\sigma_1}{\Longrightarrow} S_1 \cdots \stackrel{\alpha_t,\sigma_t}{\Longrightarrow} S_t \stackrel{\alpha_{t+1},\sigma_{t+1}}{\Longrightarrow} S_r.$$

By the induction hypothesis, each atom $A \in S_t$ is entailed by LP($\mathbf{P}$) and hence $S_t \subseteq T_{\mathrm{LP}(\mathbf{P})} \uparrow \omega$.

As all operators in $\mathcal{O}$ have empty delete lists, $S_{t+1} = S_t \cup \mathrm{Add}(\alpha_{t+1})\sigma_{t+1}$. For each atom H in the add list of α_{t+1}, there is a clause in LP($\mathbf{P}$) of the form

$$(\forall)(H \leftarrow \&_{K \in \mathrm{Pre}(\alpha_{t+1})} K).$$

The atoms in the ground conjunction $\&_{K\in \mathrm{Pre}(\alpha_{t+1})} K\sigma_{t+1}$ is a subset of S_t and hence of $T_{\mathrm{LP}(\mathbf{P})} \uparrow \omega$. As this conjunction is finite, there is an integer z such that the atoms in the ground conjunction $\&_{K\in \mathrm{Pre}(\alpha_{t+1})} K\sigma_{t+1}$ is a subset of $T_{\mathrm{LP}(\mathbf{P})} \uparrow z$. It follows, by definition of T_P, that for all $H \in \mathrm{Add}(\alpha_{t+1})$, $H\sigma_{t+1} \in T_{\mathrm{LP}(\mathbf{P})} \uparrow (z+1) \subseteq T_{\mathrm{LP}(\mathbf{P})} \uparrow \omega$. Hence, every atom in S_{t+1} is entailed by LP($\mathbf{P}$) and consequently, G is a logical consequence of LP($\mathbf{P}$). ■

Theorem 3.2 (Equivalence Theorem II) Suppose P is a definite logic program and G is any goal. Then: $P \models G$ iff there is a plan to achieve G from $\mathrm{PD}(P) = (\mathcal{S}(P), \mathcal{O}(P))$.

Proof. ($\Rightarrow$) Suppose $P \models G$. Then there is a ground instance G' of G, and a minimal integer $n > 0$ such that G' is true in $T_P \uparrow n$ (see Definition A.1 and the paragraph immediately following it for an explanation of this notation). Let $G' = (A_1 \,\&\ldots\&\, A_m)$. We proceed by induction on n.

Base Case ($n = 1$). In this case, for each $1 \leq i \leq m$, there is a clause in P having a ground instance of the form:

$$A_i \quad \leftarrow .$$

Then $\{A_1, \ldots, A_m\} \subseteq S(P)$, and hence, G' (and hence G) is true in the initial state of the planning domain PD(P).

Inductive Case ($n = k+1$): Suppose G' is true in $T_P \uparrow (k+1)$. Then, for each $1 \leq i \leq m$, there is clause C_i in P having a ground instance, $C_i\theta_i$, of the form

$$A_i \quad \leftarrow \quad B_1^i \,\&\ldots\&\, B_{w_i}^i$$

such that $\{B_1^i, \ldots, B_{w_i}^i\} \subseteq T_P \uparrow k$. Thus, the goal

$$G'' \quad = \quad \bigwedge_{i=1}^{m} \bigwedge_{\ell=1}^{w_i} B_\ell^i$$

is true in $T_P \uparrow k$. Note that G'' is a ground goal. By the induction hypothesis, there is a plan, $\mathcal{P}_0$, to achieve G''. Let W_0 be the the state obtained from the initial state $S(P)$ by executing the operations in $\mathcal{P}_0$. Then it is straightforward to see that

$$S(P) \stackrel{\mathcal{P}_0}{\Longrightarrow} W_0 \stackrel{op_{C_1},\theta_1}{\Longrightarrow} W_1 \stackrel{op_{C_2},\theta_2}{\Longrightarrow} W_2 \ldots W_{m-1} \stackrel{op_{C_m},\theta_m}{\Longrightarrow} W_m$$

is a plan that achieves G'. (Note that as each operator op_{C_i}, $1 \leq i \leq m$, is deletion-free, $W_{i+1} = W_i \cup \mathit{Add}(op_{C_i})\theta_i)$. Furthermore, as each operator, op_{C_i} is executable in W_0, it follows that it is also executable in W_i for all $1 \leq i \leq m$.) Furthermore, as $\mathit{Add}(op_{C_i})\theta_i = \{A_i\}$, it follows that $\{A_1, \ldots, A_m\} \subseteq W_m$. The above plan achieves G.

($\Leftarrow$) Suppose there is a plan to achieve G from PD(P). Then there is a ground instance $G' = (A_1 \,\&\ldots\&\, A_m)$ of G that is achievable by this plan. Let

$$S_0 \stackrel{\alpha_1,\theta_1}{\Longrightarrow} S_1 \stackrel{\alpha_1,\theta_1}{\Longrightarrow} S_2 \cdots \stackrel{\alpha_n,\theta_n}{\Longrightarrow} S_n \qquad (2)$$

be such a plan, where $S_0 = S(P)$. We proceed by induction on n, the length of the plan.

Base Case ($n = 0$). In this case, $G' = (A_1 \& \ldots \& A_m)$, is true in $S_0 = S(P)$, i.e. each A_i, $1 \leq i \leq m$, is true in $S(P)$. But then, each A_i, $1 \leq i \leq m$, is a ground instance of a fact in P. Clearly, $P \models A_i$ for all $1 \leq i \leq m$, and hence, $P \models G'$ and hence $P \models G$.

Inductive Case ($n = k + 1$). As each operator in $\mathcal{O}(P)$ contains only one element in its add list, there are exactly two possibilities: either the atom added by θ_n-execution of the operator α_n is some A_i, $1 \leq i \leq m$, or it is not. In the latter case, as S_n satisfies G', and as the last step in the plan does not cause any of the A_i's to be added, G' must be true in S_{n-1}. Hence, by the induction hypothesis, we can assume that $P \models G'$, and hence, $P \models G$. In the other case, there is an integer $1 \leq i \leq m$ such that A_i is the atom added by θ_n-executing α_n. Furthermore,

$$G^* \quad = \quad A_1 \& \ldots \& A_{i-1} \& A_{i+1} \& \ldots \& A_m$$

is true in S_n. Hence, by the induction hypothesis, as there is a plan of length $(n-1)$ to achieve G^*, it follows that $P \models G^*$. Furthermore, as α_n is θ_n-executable in state S_{n-1}, it follows that the goal

$$G^{**} \quad = \quad \bigwedge_{B \in Pre(\alpha_n)} B\theta_n$$

is true in S_{n-1}. Let C be the clause in P such that $op_C = \alpha_n$. Then the clause $C\theta_n$ is of the form:

$$A_i \quad \leftarrow \bigwedge_{B \in Pre(\alpha_n)} B\theta_n.$$

As $\bigwedge_{B \in Pre(\alpha_n)} B\theta_n$ is true in S_n, by the induction hypothesis, it follows that $P \models \bigwedge_{B \in Pre(\alpha_n)} B\theta_n$. As $C \in P$, $P \models C$, and hence $P \models C\theta_n$; thus, $P \models A_i$. We already know that $P \models G^*$; the conjunction of G^* and A_i is equivalent to G'. Consequently, $P \models G'$. As G' is a ground instance of G, it follows that $P \models G$. ■

A.2 Undecidability and Decidability Results

Corollary 3.1 (Semi-Decidability Results)

1. $\{G : G$ is an existential goal such that there is a plan to achieve G from $\mathbf{P} = (S_0, \mathcal{O})\}$ is a recursively enumerable subset of the set of all goals.

2. Given any recursively enumerable collection X of ground letion-free, then PLAN EXISTENCE is strictly semi-decidable.

Proof. Immediate consequence of Theorem 3.1 and a result of Blair which shows that any recursively enumerable set of ground atoms can be represented as the set of ground atoms provable from a logic program [4]. ■

Corollary 3.2 The problem "given a positive deletion-free planning domain $\mathbf{P} = (S_0, \mathcal{O})$, is the set of goals achievable from $\mathbf{P}$ decidable?" is Π_2^0-complete.

Proof. Immediate consequence of Theorem 3.1 and a result of Blair [4] which shows that the class of determinate logic programs ([4]) is Π_2^0-complete. ∎

Corollary 3.3 If we restrict **P** to be positive, deletion-free, and context-free, then PLAN EXISTENCE is still strictly semi-decidable.

Proof. Immediate consequence of Theorem 3.1 and a result of Sebelik and Stepanek [34] that shows that all recursively enumerable sets of ground atoms can be captured as the set of ground atomic consequences of a logic program whose rules contain at most one atom in the body. ∎

Theorem 3.3 If we restrict **P** to be deletion-free and function-free, then PLAN EXISTENCE is decidable.

Proof. As **P** contains no function symbols, and as our language has only finitely many constant and predicate symbols, the set of ground atoms in our language is finite, and hence so is the power set of this set (i.e. the set of states is finite). Furthermore, the number of ground instances of operators in our planning domain is also finite. Associate with the planning domain $\mathbf{P} = (S_0, \mathcal{O})$, a finite graph. For each state s, there is a vertex labeled s in the graph. There is an edge from the vertex labeled s to the vertex labeled s' iff there is an operator in $\mathcal{O}$ having a ground instance such that s satisfies the preconditions of the ground instance, and s' is the state obtained by applying the ground operator in state s. The graph is finite, and clearly, there is a plan to achieve a given goal G iff there is a path from S_0 to a state S_1 in which G is true. This problem is clearly decidable. ∎

Theorem 3.4 (Eliminating Delete Lists and Negated Preconditions) Suppose **P** is a function-free planning domain. Then there is a positive deletion-free planning domain $\mathbf{P}' = (S_0', \mathcal{O}')$ such that for each goal

$$G \equiv (\exists)(A_1 \,\&\ldots\&\, A_n)$$

there is a goal

$$G' \equiv (\exists)(A_1' \,\&\ldots\&\, A_n' \,\&\, \mathrm{poss}(S))$$

where "poss" is a new unary predicate symbol and for all $1 \le i \le n$, if $A_i \equiv p(t_1, \ldots, t_n)$, then $A_i' \equiv p(t_1, \ldots, t_n, S)$ where S is a variable symbol. Furthermore, G is achievable from **P** iff G' is achievable from $\mathbf{P}'$.

Proof. As $\mathcal{L}$ is function free, the set of ground atoms is finite (say k in number). Hence there are only 2^k states expressible in language $\mathcal{L}$. Extend $\mathcal{L}$ to a new language $\mathcal{L}'$ by adding the following new symbols:

1. new constant symbols $s_1, \ldots, s_{2^k}$;

2. a new unary predicate symbol "poss."

Intuitively, think of each new constant symbol s_i as representing a state, denoted $\mathrm{REP}(s_i)$, of language $\mathcal{L}$. Thus, $\mathrm{REP}(s_i)$ is a collection (finite) of ground atoms of $\mathcal{L}$. Clearly, REP is a bijection between $\{s_1, \ldots, s_{2^k}\}$ and the set of states of $\mathcal{L}$. We assume that the constant symbol s_{init}, $1 \le \mathrm{init} \le 2^k$, denotes the initial state S_0 of **P**. Construct S_0' as follows:

1. $\mathrm{poss}(s_{\mathrm{init}}) \in S_0'$.
2. For all $1 \leq i \leq 2^k$, if $A = p(t_1, \ldots, t_n) \in \mathrm{REP}(s_i)$, then $\tilde{A} = p(t_1, \ldots, t_n, s_i) \in S_0'$.
3. Nothing else is in S_0'.

Note that S_0' as defined above, only contains ground atoms in the expanded language $\mathcal{L}'$.

Now construct operators as follows: Suppose $\alpha \in \mathcal{O}$, S_i, S_j are states of $\mathcal{L}$ and θ is a ground substitution for the variables in $\mathrm{Name}(\alpha)$ such that

$$S_i \overset{\alpha,\theta}{\Longrightarrow} S_j.$$

Then the following operator is in $\mathcal{O}'$:

Pre: $\{\mathrm{poss}(s_i)\}$
Add: $\{\mathrm{poss}(s_j)\}$
Del: $\emptyset$

(Here, s_i and s_j are the constant symbols corresponding to states S_i, S_j respectively). Thus, $\mathcal{O}'$ is constructed by considering all possible combinations of $\alpha \in \mathcal{O}$, states S_i, S_j and ground substitutions for the variables in each α. As $\mathcal{L}$ is function-free (and hence contains only finitely many ground terms), $\mathcal{O}'$ is finite and contains no delete lists. It is easy to see, from the construction, that G is achievable form $\mathbf{P}$ iff G' is achievable from $\mathbf{P}'$. ∎

Theorem 3.5 (Decidability of Function-Free Planning) If we restrict $\mathbf{P}$ to be function-free, then PLAN EXISTENCE is decidable.

Proof. Immediate consequence of Theorems 3.4 and 3.3. ∎

A.3 Restricted Planning Domains

Theorem 3.7 (Decidability for Weakly Recurrent Planning Domains) If $\mathbf{P} = (S_0, \mathcal{O})$ is restricted to be weakly recurrent (via witness ℓ), positive, and deletion-free, and G is restricted to be bounded w.r.t. ℓ, then PLAN EXISTENCE is decidable.

Proof. Here's an algorithm for this purpose. Let $G = (\exists)(A_1 \& \ldots \& A_n)$. Let p_{new} be a new propositional symbol not present in $\mathbf{P}$. Introduce a new operator, α_{new} as follows:

Pre: $\{A_1, \ldots, A_n\}$
Add: $\{p_{new}\}$
Del: $\emptyset$.

Let $\mathbf{P}' = (S_0, \mathcal{O} \cup \{\alpha_{new}\})$. It is easy to see that $\mathbf{P}'$ achieves the ground goal p_{new} iff $\mathbf{P}$ achieves goal G. $\mathbf{P}'$ can clearly be effectively constructed from $\mathbf{P}$. Let ℓ be the level mapping which witness the weakly-recurrent property of $\mathbf{P}$ and let b be the integer via which G is bounded by ℓ. Extend ℓ so that $\ell(p_{new}) = b + 1$. ℓ extended in this manner witnesses the weakly-recurrent property of $\mathbf{P}'$.

Convert $\mathbf{P}'$ to $\mathrm{LP}(\mathbf{P}')$. By theorem 3.1, there is a plan to achieve p_{new} from $\mathbf{P}'$ iff $\mathrm{LP}(\mathbf{P}') \models p_{new}$. By Proposition 3.3, $\mathrm{LP}(\mathbf{P}')$ is weakly recurrent. Hence, by Theorem 3.6, there is a terminating procedure that, given a logic program Q and any bounded goal G' as input, will determine whether $Q \models G'$. Apply this terminating procedure with inputs $\mathrm{LP}(\mathbf{P}')$ and $G' = p_{new}$. If the procedure terminates with *yes*, then goal G is achievable from $\mathbf{P}$, whereas if it terminates with a *no*, G is not achievable. ∎

A.4 Extended Planning Domains

Theorem 3.8 If $\mathbf{P} = (S_0, \mathcal{O})$ is restricted to be positive and deletion-free, but the initial state S_0 is allowed to be an infinite, decidable set of ground atoms, PLAN EXISTENCE is strictly semi-decidable.

Proof. It is easy to see that each $T_{\mathrm{LP}(\mathbf{P})} \uparrow n$ is decidable. Thus, $T_{\mathrm{LP}(\mathbf{P})} \uparrow \omega$ is semi-decidable as $A \in T_{\mathrm{LP}(\mathbf{P})} \uparrow \omega$ iff $(\exists n < \omega) A \in T_{\mathrm{LP}(\mathbf{P})} \uparrow n$. As there exists a plan that achieves goal G from $\mathbf{P}$ iff there exists an $n < \omega$ such that G is true in $T_{\mathrm{LP}(\mathbf{P})} \uparrow n$, it follows that the problem at hand is semi-decidable. ■

Theorem 3.9 Any planning domain $\mathbf{P} = (S_0, \mathcal{O})$ whose language $\mathcal{L}$ contains function symbols can be reduced to an equivalent planning domain $\mathbf{P}' = (S_0', \mathcal{O}')$ whose language $\mathcal{L}'$ contains infinitely many constants but no function symbols, provided that we allow the initial state S_0' to be infinite. $\mathbf{P}'$ is equivalent to $\mathbf{P}$ in the sense that for every goal G in $\mathbf{P}$, there is a goal G' in $\mathbf{P}'$ such that there is a plan for G in $\mathbf{P}$ iff there is a plan for G' in $\mathbf{P}'$.

Proof. The reduction is as follows:

• $\mathcal{L}'$ contains all of the constant symbols, variable symbols, and predicate symbols found in $\mathcal{L}$. For each term t in $\mathcal{L}$ that is not a constant or variable symbol, $\mathcal{L}'$ contains a new constant symbol c_t. For each n-ary function symbol f in $\mathcal{L}$, $\mathcal{L}'$ contains a new $n+1$-ary predicate symbol e_f.

• Let t be any term, atom, or negated atom in $\mathcal{L}$. Then t's *translation* $T(t)$ and *equivalence conditions* $E(t)$ are defined as follows:

1. If s is a predicate symbol, constant symbol, or variable symbol, then $T(s) = s$, and $E(s) = \emptyset$.
2. Suppose t is a term of the form $f(t_1, \ldots, t_n)$. Then
$$\begin{aligned} T(t) &= k_t; \\ E(t) &= \{e_f(k_t, T(t_1), T(t_2), \ldots, T(t_n))\} \cup E(t_1) \cup \ldots \cup E(t_n). \end{aligned}$$
3. Suppose $t = p(t_1, \ldots, t_n)$, where p is a predicate symbol, and $t_1, t_2, \ldots, t_n$ are terms. Then
$$\begin{aligned} T(t) &= p(k_{t_1}, k_{t_2}, \ldots, k_{t_n}); \\ E(t) &= E(t_1) \cup \ldots \cup E(t_n). \end{aligned}$$
4. The only other possibility is that $t = \neg p(t_1, \ldots, t_n)$, where p is a predicate symbol, and $t_1, t_2, \ldots, t_n$ are terms. In this case,
$$\begin{aligned} T(t) &= \neg p(k_{t_1}, k_{t_2}, \ldots, k_{t_n}); \\ E(t) &= E(t_1) \cup \ldots \cup E(t_n). \end{aligned}$$

Below are two examples:

1. If a is the atom $p(f(x,d), g(b), z, c)$ then

$$\begin{aligned} T(a) &= p(k_{f(x,d)}, k_{g(b)}, y, c); \\ E(a) &= \{e_f(x, d, k_{f(x,d)}), e_g(b, k_{g(b)})\}. \end{aligned}$$

2. If a is the atom $a = \neg p(g(f(x), y), h(c), z)$, then

$$\begin{aligned} T(a) &= \neg p(k_{g(f(x),y)}, k_{h(c)}, y); \\ E(a) &= \{e_g(k_{f(x)}, y, k_{g(f(x),y)}), e_f(x, k_{f(x)}), e_h(c, k_{h(c)}). \end{aligned}$$

• In $\mathbf{P}'$, the initial state S_0' contains the translations and equivalence conditions for all atoms in S_0, plus the equivalence conditions for all terms in $\mathcal{L}$.

• To translate planning operators, we remove function symbols, and include the corresponding equivalence conditions as preconditions. More specifically, let O be any planning operator in $\mathcal{O}$. Then the translation O' of O is as follows:

1. If Name$(O) = O(x_1, \ldots, x_n)$, then Name$(O') = O'(x_1, \ldots, x_n)$.
2. Pre(O') contains the translations and equivalence conditions of all atoms in Pre(O), plus the equivalence conditions for all atoms in Add(O) and Del(O).
3. Add(O') contains the translations of all atoms in Add(O).
4. Del(O') contains the translations of all atoms in Del(O).

• Using the above, it is easy to show that

$$S_0 \overset{\alpha_1,\theta_1}{\Longrightarrow} S_1 \overset{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \overset{\alpha_n,\theta_n}{\Longrightarrow} S_n$$

is a plan in $\mathbf{P}$ that achieves some goal G, iff

$$S_0' \overset{\alpha_1',\theta_1}{\Longrightarrow} S_1' \overset{\alpha_2',\theta_2}{\Longrightarrow} S_2' \cdots \overset{\alpha_n',\theta_n}{\Longrightarrow} S_n'$$

is a plan in $\mathbf{P}'$ that achieves $\{T(a) : a \in G\}$. ■

Theorem 3.10 If $\mathbf{P}$'s language $\mathcal{L}$ is allowed to contain infinitely many constants, then PLAN EXISTENCE is semi-decidable even if $\mathbf{P} = (S_0, \mathcal{O})$ is restricted to be deletion-free and function-free (and S_0 is finite).

Proof. Here, we show that given any deterministic TM M, we can encode it as a deletion-free planning problem with infinitely many constants.

The TM is denoted by $M = (K, \Sigma, \Gamma, \#, \delta, q_0, F)$. $K = \{q_0, \ldots, q_m\}$ is a finite set of states. $F \subseteq K$ is the set of so called final states. Γ is the finite set of allowable tape symbols. $\Sigma \subseteq \Gamma$ is the set of allowable input symbols. $\#$ is the blank tape symbol. $q_0 \in K$ is the start state. δ is the next move function.

Suppose we are given a TM M, and an input string $x = (x_0, x_1, \ldots, x_n)$ such that $x_i \in \Sigma$ for each i. We can map this into the following planning problem :

Constant symbols: We have an infinite number of constant symbols. Some of them are designated to denote the states of the Turing machine, the tape symbols, the tape cells, the steps of the Turing machine.

Predicates:
done() is a propositional predicate denoting that the goal is achieved.
state(t, s) is used to denote that the machine is in state s at step t.
contains(t, j, x) is used to denote that at step t, the tape cell j contains the symbol x.
head(t, j) is used to denote that at step t, the head is at cell j.
cell(j) is used to denote that j is a tape cell.
right(j, k) is used to denote that cell k is to the right of cell j.
same(j, k) is used to denote that j and k refer to the same cell.
used(j) is a predicate used to create new cells.
oldstep(j) is used to keep track of steps.
next(t, t') is used to denote that t' is the step that follows t.

Initial State:
state$(0, s_0)$, head$(0, 0)$,
contains$(0, 0, x_0)$,..., contains$(0, n, x_n)$
right$(0, 1)$,...,right$(n-1, n)$,
cell(0),..., cell(n),
same$(0, 0)$,..., same(n, n),
used(0),..., used$(n-1)$, oldstep(0)

Operators: For each $q \in F$, we have the operator

Pre:	$\{\mathrm{state}(V, q)\}$
Del:	$\emptyset$
Add:	$\{\mathrm{done}()\}$

Whenever $\delta(q, a) = (q', b, Right)$, we have the following two operators.

Pre:	$\{\mathrm{state}(T, q), \mathrm{head}(T, J), \mathrm{contains}(T, J, a), \mathrm{right}(J, J'), \neg\mathrm{oldstep}(T')\}$
Del:	$\emptyset$
Add:	$\{\mathrm{state}(T', q'), \mathrm{head}(T', J'), \mathrm{contains}(T', J, a'), \mathrm{next}(T, T'), \mathrm{oldstep}(T')\}$

Pre:	$\{\mathrm{contains}(T, J, X), \mathrm{head}(T, J'), \neg\mathrm{same}(J, J'), \mathrm{next}(T, T')\}$
Del:	$\emptyset$
Add:	$\{\mathrm{contains}(T', J, X)\}$

Whenever $\delta(q, a) = (q', b, Left)$, we have two other operators that are obtained by replacing right(J, J') with right(J', J) in the two operators above.

Here is the operator used to create new tape cells:

Pre:	$\{\mathrm{cell}(V), \neg\mathrm{used}(V), \neg\mathrm{cell}(V')\}$
Del:	$\emptyset$
Add:	$\{\mathrm{used}(V), \mathrm{cell}(V'), \mathrm{right}(V, V'), \mathrm{contains}(0, V', \#), \mathrm{same}(V', V')\}$

The planner simulates the Turing machine, move for move. There exists a plan for done() iff the Turing machine halts. Hence planning in a deletion-free domain with infinitely many constants but a final initial state is undecidable. ■

Definition A.2 A constant that appears in the initial state or in an operator definition is called a *basic constant.* All other constants are called *non-basic constants.*

Lemma A.2 *Let* $\mathbf{P} = (S_0, \mathcal{O})$ *be a function-free, deletion-free, positive planning domain such that the planning language* $\mathcal{L}$ *contains infinitely many constants (but* S_0*, as usual, is finite). For every goal* G*, there is a plan in* $\mathbf{P}$ *that achieves* G *iff there is a plan in* $\mathbf{P}$ *with finitely many constants that achieves* G*. Furthermore, this finite set of constants contains only those constants found in the initial state or the operator definitions, if any (otherwise, it contains a single constant).*

Proof. ($\Leftarrow$). This direction is trivial.

($\Rightarrow$). Assume there exists a plan that uses possibly non-basic constants. Pick up arbitrarily, one of the basic constants, if there exists any. Otherwise, pick any one constant. Let us call the constant we picked as a.

In the plan, replace all non-basic constants with a. We get a valid plan that achieves the goal. Here is why.

Recall that **P**is positive and deletion free. Hence, anything asserted remains asserted, and all preconditions are positive. Consider a precondition of any operator or any goal. Since the original plan is valid, either there is a previous operator that asserts it, or it is true in the initial state. If there is a previous operator that asserts it, then the same operator in the modified plan will assert the modified precondition/goal. If it is true in the initial state, then it did not contain any non-basic variables, hence, it was not affected by the replacement. It should be satisfied in the modified plan, too. ■

Theorem 3.11 If the language is allowed to contain infinitely many constants but $\mathbf{P} = (S_0, \mathcal{O})$ is restricted to be positive, deletion-free, and function-free (and S_0 is finite), then PLAN EXISTENCE is decidable.

Proof. Direct consequence of Lemma A.2 and Theorem 3.5. ■

B Proofs of Complexity Results

B.1 Binary Counters

Several of the proofs in this paper depend on using function-free ground atoms to represent binary n-bit counters, and function-free planning operators to increment and decrement these counters. Below, we show how this can be done.

To represent a counter that can be incremented, we would like to have an atom $c(i)$ whose intuitive meaning is that the counter's value is i, and an operator "incr" that deletes $c(i)$ and replaces it by $c(i+1)$, respectively. The problem is that without function symbols, we cannot directly represent the integer i nor the arithmetic operation on it. However, since we have the restriction $0 \leq i \leq 2^n - 1$ for some n, then we can achieve the same effect by encoding i in binary as

$$i = i_1 \times 2^{n-1} + i_2 \times 2^{n-2} + \ldots + i_{n-1} \times 2^1 + i_n,$$

where each i_k is either 0 or 1. Instead of the unary predicate $c(i)$, we can use an n-ary predicate $c(i_1, i_2, \ldots, i_n)$; and to increment the counter we can use the following operators:

Name: $\text{incr}_1(i_1, i_2, \ldots, i_{n-1})$
Pre: $\{c(i_1, i_2, \ldots, i_{n-1}, 0)\}$
Del: $\{c(i_1, i_2, \ldots, i_{n-1}, 0)\}$
Add: $\{c(i_1, i_2, \ldots, i_{n-1}, 1)\}$

Name: $\text{incr}_2(i_1, i_2, \ldots, i_{n-2})$
Pre: $\{c(i_1, i_2, \ldots, i_{n-2}, 0, 1)\}$
Del: $\{c(i_1, i_2, \ldots, i_{n-2}, 0, 1)\}$
Add: $\{c(i_1, i_2, \ldots, i_{n-2}, 1, 0)\}$

$\vdots$

Name: $\text{incr}_n()$
Pre: $\{c(0, 1, 1, \ldots, 1)\}$
Del: $\{c(0, 1, 1, \ldots, 1)\}$
Add: $\{c(1, 0, 0, \ldots, 0)\}$

For each $i < 2^n - 1$, exactly one of the incr_j will be applicable to $c(i_1, i_2, \ldots, i_n)$, and it will increment i by one. If we also wish to decrement the counter, then similarly we can define a set of operators $\{\text{decr}_k : k = 1, \ldots, n\}$ as follows:

Name: $\text{decr}_k(i_1, i_2, \ldots, i_{n-k+1})$
Pre: $\{c(i_1, i_2, \ldots, i_{n-k+1}, 1, 0, \ldots, 0)\}$
Del: $\{c(i_1, i_2, \ldots, i_{n-k+1}, 1, 0, \ldots, 0)\}$
Add: $\{c(i_1, i_2, \ldots, i_{n-k+1}, 0, 1, \ldots, 1)\}$

For each $i > 0$, exactly one of the decr_k will be applicable to $c(i_1, i_2, \ldots, i_n)$, and it will decrement i by one.

Suppose we want to have two n-bit counters having values $0 \leq i \leq 2^n$ and $0 \leq j \leq 2^n$, and an operator that increments i and decrements j simultaneously. If we represent the counters by n-ary predicates $c(i_1, i_2, \ldots, i_n)$ and $d(j_1, j_2, \ldots, j_n)$, then we can simultaneously increment i and decrement j using a set of operators $\{\text{shift}_{hk} : h = 1, 2, \ldots, n,\ k = 1, 2, \ldots, n\}$ defined as follows:

Name: $\text{shift}_{hk}(i_1, i_2, \ldots, i_{n-h+1}, j_1, j_2, \ldots, j_{n-k+1})$
Pre: $\{c(i_1, i_2, \ldots, i_{n-h+1}, 0, 1, 1, \ldots, 1), d(j_1, j_2, \ldots, j_{n-k+1}, 1, 0, 0, \ldots, 0)\}$
Del: $\{c(i_1, i_2, \ldots, i_{n-h+1}, 0, 1, 1, \ldots, 1), d(j_1, j_2, \ldots, j_{n-k+1}, 1, 0, 0, \ldots, 0)\}$
Add: $\{c(i_1, i_2, \ldots, i_{n-h+1}, 1, 0, 0, \ldots, 0), d(j_1, j_2, \ldots, j_{n-k+1}, 0, 1, 1, \ldots, 1)\}$.

For each i and j, exactly one of the shift_{hk} will be applicable, and it will simultaneously increment i and decrement j.

For notational convenience, instead of explicitly defining a set of operators such as the set $\{\text{incr}_h : h = 1, \ldots, n\}$ defined above, we sometimes will informally define a single "abstract operator" such as

Name: $\text{incr}(\underline{i})$
Pre: $\{c(\underline{i})\}$
Del: $\{c(\underline{i})\}$
Add: $\{c(\underline{i+1})\}$

where $\underline{i}$ is the sequence $i_1, i_2, \ldots, i_n$ that forms the binary encoding of i. Whenever we do this, it should be clear from context how a set of actual operators could be defined to manipulate $c(i_1, i_2, \ldots, i_n)$.

B.2 Eliminating Negated Preconditions

Theorem 5.1 (Eliminating Negated Preconditions) In polynomial time, given any planning domain $\mathbf{P} = (S_0, \mathcal{O})$ we can produce a positive planning domain $\mathbf{P}' = (S_0', \mathcal{O}')$ having the following properties:

1. For every goal G, a plan exists for G in $\mathbf{P}$ if and only if a plan exists for G in $\mathbf{P}'$.
2. For every goal G and non-negative integer l, there exists a plan of length l for G in $\mathbf{P}$ if and only if there exists a plan of length $l + 2^{kv}$ for G in $\mathbf{P}'$, where k is the maximum arity among the predicates of $\mathbf{P}$ and $v = \lceil \lg c \rceil$, where c is the number of constants in $\mathbf{P}$ (i.e., v is the number of bits necessary to encode the constants in binary).

Proof. Here is the transformation:

Predicates: $P' = P \cup \{p' | p \in P\} \cup \{\text{counter}, \text{start}\}$

Intuitively, p' is the complementary predicate for p. That is, whenever the ground atom $p(\ldots)$ is true, $p'(\ldots)$ is false. Without loss of generality, we assume all predicates in P have the same arity. This can be achieved by adding dummy arguments to some of the predicates; we modify G and S_0 so that these dummy arguments have fixed values. Furthermore, we use {0,1} as our set of constants; this can easily be achieved by encoding each constant as a binary string of ones and zeroes, and increasing the number of arguments to the predicates by v.

counter$(\ldots)$ kv-ary.

start() is 0-ary (i.e., it is a proposition).

Initial state: $\{\text{counter}(\underline{0})\}\{p'(\underline{0}) : p \in P\}$

Goal state: G

Operators: For each operator $O \in \mathcal{O}$, we have the following operator $O' \in \mathcal{O}'$ that imitates it:

Pre:	$S_1 \cup S_2 \cup \{\text{start}()\}$
Del:	$\text{Del}(O) \cup \{p' \mid p \in \text{Add}(O)\}$
Add:	$\text{Add}(O) \cup \{p' \mid p \in \text{Del}(O)\}$

where S_1 is the set of all nonnegated atoms in $\text{Pre}(O)$, and S_2 is the set of complementary predicates corresponding to the negated atoms in $\text{Pre}(O)$.

The idea is to replace the negative literals in the precondition list with complementary predicates. Whenever we add a predicate instance, we delete its complementary predicate instance, and whenever we delete a predicate instance, we add its complementary predicate instance.

We have the following two operators to reach the state corresponding to the initial state of the original planning problem. Increments and decrements (such as mapping $\underline{i}$ to $\underline{i+1}$ or $\underline{i-1}$) should be handled as described in Section B.1.

Pre:	$\{\text{counter}(\underline{i})\}$
Del:	$\{\text{counter}(\underline{i})\}$
Add:	$\{\text{counter}(\underline{i+1})\} \cup \{p'(\underline{i+1}) : p \in P\}$

Pre:	$\{\text{counter}(\underline{2^{kv}-1})\}$
Del:	$\{\text{counter}(\underline{2^{kv}-1})\} \cup \{p'(\underline{j}) : p(\underline{j}) \in S_0\}$
Add:	$\{p(\underline{j}) : p(\underline{j}) \in S_0\} \cup \{\text{start}()\}$

In the first 2^{kv} steps of any plan in $\mathbf{P}'$, start() would be false. These steps are used to assert the instances of complementary predicates. Then, we start imitating the original planning problem move to move. Hence if there exists a plan of length l in the original planning problem, there exists a plan of length $l + 2^{kv}$ in this planning problem. The transformation is obviously polynomial.

■

B.3 Planning When the Operator Set is Part of the Input

Theorem 5.3 If we restrict $\mathbf{P}$ to be propositional, positive, context-free, and deletion-free, then PLAN EXISTENCE is NLOGSPACE-complete.

Proof. Below, we show that the problem is in NLOGSPACE and that it is NLOGSPACE-hard.

Membership. Here is an NLOGSPACE algorithm that decides this problem:

1. For each proposition p in G do:
   (a) $g := p$
   (b) if g is in the initial state, continue with the next proposition in G.
   (c) Nondeterministically choose an operator with g in the addlist. If no such operator exists, halt and reject.
   (d) $g :=$ the precondition of the operator if it exists, *TRUE* otherwise.
   (e) Go to Step 1(b).
2. Halt and accept.

The algorithm is based on two facts: Since $\mathbf{P}$ is restricted to be positive and deletion-free, the subgoals do not interact. Hence we can look for a plan for each of them separately. Secondly, $\mathbf{P}$ is restricted to be context-free, that is each operator has at most one precondition. As a result, in step (b), we do not need to consider multiple preconditions.

The algorithm accepts iff there exists a plan that achieves the goal. Only space required is for g, and for keeping track of the iteration in the for loop. Hence the problem is in NLOGSPACE.

Hardness. In order to complete the proof, we give a logspace reduction from off-line logspace-bounded nondeterministic TM acceptance problem to the propositional planning problem with the previous restrictions.

An off-line logspace-bounded TM is defined as an n-tuple $M = (\ldots)$. Basically, it is a Turing machine with one read-only input tape, one write only output tape, and a read/write work tape. The head of the output tape can not move left. Given input x in the input tape, it uses at most $\lceil \lg |x| \rceil$ cells on the work tape. A configuration of the TM can be represented with the positions of the three heads, the current state, and the contents of the work tape.

We do not need to include the contents of the output tape since we can not read it anyway, and we do not need the contents of the input tape explicitly as it never changes.

Given an off-line logspace-bounded nondeterministic TM, and input x, the number of possible configurations is polynomial in terms of the input. Hence, a configuration of M can be encoded in logarithmic space. We introduce a proposition for each of these configurations. We enumerate all these configurations and for each of them we output operators such that precondition list contains the proposition corresponding to the configuration, and the addlist contains a proposition corresponding to a configuration reachable from the configuration in the precondition via some move. In addition to these, we create an operator for each halting configuration such that the precondition contains the proposition corresponding to it, and the addlist contains a special proposition called *done*, which will also be the goal. Note that these can be done in NLOGSPACE.

The TM will accept x iff there exists a plan that achieves done, starting from proposition S_0, which corresponds to the initial configuration of the TM. ∎

Theorem 5.4 If we restrict **P** to be propositional, positive, context-free and deletion-free, then PLAN LENGTH is NP-complete.

Proof. Since we do not have any delete lists, any operator need to appear in a plan at most once. Number of operators is bounded by the length of the input. Hence we can nondeterministically guess a sequence of operators and verify that the sequence is a plan of length at most k, in polynomial time. Therefore, the problem is in NP.

The Set Cover problem, defined defined below, is known to be NP-complete [16].

> Given a set S, a set C which is a collection of subsets of S, and a positive integer k encoded in binary, is there a subset $C' \subseteq C$ of size at most k, such that each element of S appears in some set in C'?

To prove that our planning problem is NP-hard, we define the following polynomial-time reduction from the Set Cover problem:

Propositions: For each element a of S, we have a proposition p_a.

Operators: For each subset $\{a_1, a_2, \ldots, a_m\} \in C$, we have the following operator:

Pre: $\emptyset$
Add: $\{p_{a1}, p_{a2}, \ldots, p_{am}\}$
Del: $\emptyset$

Initial state: $\emptyset$

Goal state: $\{p_a | a \in S\}$

S has a set cover of size at most k, iff there exists a plan of size at most k. The reduction is obviously polynomial. Note that all the operators are context-free,deletion-free and positive. ∎

Theorem 5.5 PLAN LENGTH is PSPACE-complete if we restrict **P** to be propositional. It is still PSPACE-complete if we restrict **P** to be propositional and positive.

Proof.

Membership. Since **P** is restricted to be propositional, the size of any planning state will not exceed number of propositions. Hence any state can be represented in polynomial space.

The following algorithm solves the problem in NPSPACE. Starting with the initial state, we nondeterministically choose an operator, apply it to get the next state, and decrement k. We repeat this until we find a plan in which case we accept, or until $k = 0$, in which case we reject. Since PSPACE equals NPSPACE, the problem is in PSPACE.

Hardness. The existence version of this problem has been shown to be PSPACE-complete. (Theorem 5.2) We can reduce it to our problem, just by setting $k = 2^n$, where n is the number of propositions. Notice that k will be encoded in n bits. If there exists a plan, there also exists a plan of length no more than k, because the number of states in the planning problem is exponential in terms of number of propositions. This completes the proof that our problem is PSPACE-complete. ■

Theorem 5.6 (Composition Theorem) Let $\mathbf{P} = (S_0, \mathcal{O})$ be a planning domain, and $\mathcal{O}'$ be a set of operators such that each operator in $\mathcal{O}'$ is the composition of operators in $\mathcal{O}$. Then for any goal G, there is a plan to achieve G in **P** iff there is a plan to achieve G in $\mathbf{P}'$, where $\mathbf{P}' = (S_0, \mathcal{O} \cup \mathcal{O}')$.

Proof. Since operators in $\mathcal{O}'$ are compositions of operators in $\mathcal{O}$, any plan that contains operators from $\mathcal{O}'$ can be expressed without these operators, just by replacing each occurrence of operators from $\mathcal{O}'$, by the sequence of operators in $\mathcal{O}$, whose composition gives these operators.

Thus, there exists a plan to achieve G in **P** iff there exists a plan to achieve G in $\mathbf{P}'$. ■

Theorem 5.7 PLAN EXISTENCE is EXPSPACE-complete. It is still EXPSPACE-complete if we restrict **P** to be positive.

Proof. Below, we show that the problem is in EXPSPACE and that it is EXPSPACE-hard.

Membership. The number of ground instances of predicates involved is exponential in terms of the input length. Hence the size of any state can not be more than exponential. Starting from the initial state, we nondeterministically choose an operator and apply it. We do this repeatedly until we reach the goal, solving the planning problem in NEXPSPACE. NEXPSPACE is equal to EXPSPACE, hence our problem is in EXPSPACE.

Hardness. To complete the proof, we define a polynomial reduction from the EXPSPACE-bounded TM problem, which is defined as follows:

> Given a TM M that uses at most an exponential number of tape cells in terms of the length of its input, and an input string x, does M accept the string x?

A Turing machine M is normally denoted by $M = (K, \Sigma, \Gamma, \delta, q_0, F)$. $K = \{q_0, \ldots, q_m\}$ is a finite set of states. $F \subseteq K$ is the set of final states. Γ is the finite set of allowable tape symbols. $\Sigma \subseteq \Gamma$ is the set of allowable input symbols. $q_0 \in K$ is the start state. δ, the next move function, is a mapping from $K \times \Gamma$ to $K \times \Gamma \times \{\text{Left}, \text{Right}\}$

Suppose we are given M, and an input string $x = (x_0, x_2, \ldots, x_{n-1})$ such that $x_i \in \Sigma$ for each i. To map this into a planning problem, the basic idea is to represent the machine's current state, the location of the head on the tape, and the contents of the tape by a set of atoms.

The transformation is as follows:

Predicates: contains$(\underline{i}, c)$ means that c is in the i'th tape cell, where $\underline{i} = i_1, i_2, \ldots, i_n$ is the binary representation of i. We can write c on cell i by deleting contains$(\underline{i}, d)$ and adding contains$(\underline{i}, c)$, where d is the symbol previously contained in cell i.

state(q) means that the current state of the TM is q.

$h(\underline{i})$ means that the current head position is i. We can move the head to the right or left by deleting $h(\underline{i})$, and adding $h(\underline{i+1})$ or $h(\underline{i-1})$.

counter$(\underline{i})$ is a counter for use in initializing the tape with blanks.

start() denotes that initialization of the tape has been finished.

Constant symbols: $\Gamma \cup K \cup \{0, 1\}$

Operators: Each operator below that contains increment or decrement operations (such as mapping i to $i+1$ or $i-1$) should be expanded into n operators as described in Section B.1.

Whenever $\delta(q, c)$ equals (s, c', Left), we create the following operator:

Name:	$L_{q,c}^{s,c'}(\underline{i})$
Pre:	$\{h(\underline{i}), \text{state}(q), \text{contains}(\underline{i}, c), \text{start}()\}$
Del:	$\{h(\underline{i}), \text{state}(q), \text{contains}(\underline{i}, c)\}$
Add:	$\{h(\underline{i-1}), \text{state}(s), \text{contains}(\underline{i}, c')\}$

Whenever $\delta(q, c)$ equals (s, c', Right), we create the following operator:

Name:	$R_{q,c}^{s,c'}(\underline{i})$
Pre:	$\{h(\underline{i}), \text{state}(q), \text{contains}(\underline{i}, c), \text{start}()\}$
Del:	$\{h(\underline{i}), \text{state}(q), \text{contains}(\underline{i}, c)\}$
Add:	$\{h(\underline{i+1}), \text{state}(s), \text{contains}(\underline{i}, c')\}$

We have the following operator that initializes the tape with blank symbols:

Name:	$I(\underline{i})$
Pre:	$\{\text{counter}(\underline{i}), \neg\text{start}()\}$
Del:	$\emptyset$
Add:	$\{\text{counter}(\underline{i+1}), \text{contains}(\underline{i}, \#)\}$

The following operator ends the initialization phase.

Pre:	$\{\text{counter}(\underline{2^n - 1}), \neg\text{start}()\}$
Del:	$\emptyset$
Add:	$\{\text{contains}(\underline{2^n - 1}, \#), \text{start}()\}$

Finally, for each $q \in F$ we have the operator

Name:	$F_q()$
Pre:	$\{\text{state}(q)\}$
Del:	$\emptyset$
Add:	$\{\text{done}()\}$

Initial state: $\{\text{counter}(\underline{n}), \text{state}(q_0), h(\underline{0})\} \cup \{\text{contains}(\underline{i}, x_i) : i = 0, \ldots, n-1\}$

Goal condition: done().

The transformation is polynomial both in time and space. It directly mimics the behavior of the TM. This ends the proof that planning with delete lists is EXPSPACE complete. ■

Theorem 5.8 If we restrict **P** to be deletion-free, then PLAN EXISTENCE is NEXPTIME-complete.

Proof. Below, we show that the problem is NEXPTIME and that it is NEXPTIME-hard.

Membership. Since we do not have delete lists, the instances of predicates true in a state grow monotonically during the plan, hence no instance of an operator needs to be used more than once. Besides we have only an exponential number of operator instances in terms of the length of the input. We can nondeterministically guess a sequence of operator instances (of length at most exponential) and check whether it is a plan that satisfies our goal. Hence the problem is in NEXPTIME.

Hardness. Next, we show that given any nondeterministic TM M that halts in at most exponential steps in terms of its input, we can encode it in polynomial time as a deletion-free planning problem.

The TM is denoted by $M = (K, \Sigma, \Gamma, \#, \delta, q_0, F)$. $K = \{q_0, \ldots, q_m\}$ is a finite set of states. $F \subseteq K$ is the set of so called final states. Γ is the finite set of allowable tape symbols. $\Sigma \subseteq \Gamma$ is the set of allowable input symbols. $\#$ is the blank tape symbol. $q_0 \in K$ is the start state. $\delta \subseteq (K \times \Gamma) \times (K \times (\Gamma - \#)) \times \{\text{Left}, \text{Right}\}$. δ is the next move relation.

Suppose we are given a nondeterministic TM M that runs in exponential time, and an input string $x = (x_0, x_1, \ldots, x_{n-1})$ such that $x_i \in \Sigma$ for each i. Note that M runs for at most 2^n steps. We can map this into the following planning problem :

Constant symbols: $\Gamma \cup K \cup \{0, 1\}$

Predicates: done() is a propositional predicate denoting that the goal is achieved.

counter$_1(\ldots)$ and counter$_2(\ldots)$ are n-ary predicates used as binary counters. Recall that n is the length of the input string x.

start$(\ldots)$ is an n-ary predicate used to denote that the i'th step of the TM is being simulated.

same$(\ldots)$ is a $(2n)$-ary predicate used to denote that the first n bits and the second n bits encode the same numbers.

state$(\ldots)$ is a $(n+1)$-ary predicate. The first n bits encode the step, the last place holds the current state at that step.

$h(\ldots)$ is a $(2n)$-ary predicate, the first n bits encode the step, and the second n bits encode the head position at that step.

contains$(\ldots)$ is a $(2n+1)$-ary predicate, the first n bits encode the step, the second n bits encode the cell number, and the last place holds the contents of the cell at that step.

Operators: Each operator below that contains increment or decrement operations (such as mapping $\underline{i}$ to $\underline{i+1}$ or $\underline{i-1}$) should be expanded into n operators as described in Section B.1.

For each $q \in F$, we have the operator

Name:	$\text{final}(V)$
Pre:	$\{\text{state}(V, q)\}$
Del:	$\emptyset$
Add:	$\{\text{done}()\}$

The following operator asserts the "same" predicates.

Name:	$S(\underline{i})$
Pre:	$\{\text{counter}_2(\underline{i})\}$
Del:	$\emptyset$
Add:	$\{\text{counter}_2(\underline{i+1}), \text{same}(\underline{i+1}, \underline{i+1})\}$

The following operator writes blank symbols at the end of the input string. Notice we need to go up to cell $2^n - 1$ only, because M runs in NEXPTIME, and it can not access the remaining cells.

Name:	$W(\underline{i})$
Pre:	$\{\text{counter}_1(\underline{i})\}$
Del:	$\emptyset$
Add:	$\{\text{counter}_1(\underline{i+1}), \text{contains}(\underline{0}, \underline{i+1}, \#)\}$

The following operator creates the initial configuration of M after the blank symbols have been written, and the "same" predicates have been asserted:

Pre:	$\{\text{counter}_1(\underline{2^n-1}), \text{counter}_2(\underline{2^n-1})\}$
Del:	$\emptyset$
Add:	$\{\text{state}(\underline{0}, q_0), \text{head}(\underline{0}, \underline{0}), \text{contains}(\underline{0}, \underline{0}, x_0), \ldots, \text{contains}(\underline{0}, \underline{n-1}, x_{n-1})\}$

Whenever $\delta(q, a)$ contains (q', b, t) where t is left or right, we have the following two operators.

The first operator makes the nondeterministic choice and changes the content of the current cell, the state, and the head position

Name:	$N_{q,a}^{q',a'}(\underline{i}, \underline{j})$
Pre:	$\{\text{state}(\underline{i}, q), h(\underline{i}, \underline{j}), \text{contains}(\underline{i}, \underline{j}, a), \neg\text{start}(\underline{i})\}$
Del:	$\emptyset$
Add:	$\{\text{state}(\underline{i+1}, q'), \text{head}(\underline{i+1}, \underline{j+d}), \text{contains}(\underline{i+1}, \underline{j}, b), \text{start}(\underline{i})\}$

d is +1 if t is right, and it is -1 if t is left.

The second operator copies the remaining cells in step i to step $i+1$.

Name:	$C(\underline{i}, \underline{j}, \underline{V_1}, V_2)$
Pre:	$\{\text{contains}(\underline{i}, \underline{V_1}, V_2), h(\underline{i}, \underline{j}), \neg\text{same}(\underline{j}, \underline{V_1}), \text{start}(\underline{i})\}$
Del:	$\emptyset$
Add:	$\{\text{contains}(\underline{i+1}, \underline{V_1}, V_2)\}$

Notice that all the cells are not necessarily copied before continuing with the next step. However, as soon as the head position is at one of the not copied cells, all operators are disabled, except those that copy the cells. Hence this does not cause any problem.

Initial state: $\{\text{counter}_1(\underline{n-1}), \text{counter}_2(\underline{0}), \text{same}(\underline{0}, \underline{0})\}$

Goal condition: done().

The operators in the planning problem directly mimic the behavior of M. Furthermore, the transformation can be produced in polynomial time. Thus, planning with no delete lists, but negation is NEXPTIME-complete. ■

Theorem 5.9 If we restrict **P** to be positive and deletion-free, then PLAN EXISTENCE is EXPTIME-complete.

Proof.

Membership. Since the planning domain does not have delete lists and negated preconditions, any operator whose precondition list is satisfied remains so throughout the plan. Furthermore, no operator instance needs to appear in a plan more than once. Starting with the initial state, we can iteratively choose an unused operator instance whose precondition list is satisfied, and append it to our plan. We do this until either the current state satisfies the goal in which case we accept and halt, or no such operator remains in which case we halt and reject. The number of instances for an operator is c^m, where c is the number of constants in the domain, and m is the number of variables appearing in the operator. Hence there are only an exponential number of operator instances in terms of the input length, and the algorithm halts in exponential time. Thus, the problem is in EXPTIME.

Hardness. An ATM is normally denoted by $M = (K, \Sigma, \Gamma, \#, \delta, q_0, U)$. $K = \{k_1, \ldots, k_m\}$ is a finite set of states. $U \subseteq K$ is the set of so called universal states. Other states are called existential states. Γ is the finite set of allowable tape symbols. $\Sigma \subseteq \Gamma$ is the set of allowable input symbols. $\#$ is the blank tape symbol. $q_0 \in K$ is the start state. $\delta \subseteq (K \times \Gamma) \times (K \times (\Gamma - \#)) \times \{L, R, S\}$. δ is the next move relation, where L, R, S mean "left", "right", and "stationary", respectively. A configuration of ATM consists of the contents of the non-blank portion of the tape, the current state, and the head position, denoted by the triple (s, q, j). A configuration is an accepting configuration if one of the following holds:

- The state of the configuration is a universal state with no possible moves.
- The state of the configuration is an existential state, and there exists a move in δ that leads the configuration to an accepting configuration.

- The state of the configuration is a universal state and all moves lead to an accepting configuration.

This is a recursive definition, and the first bullet provides the base case.

A Linearly Bounded ATM (LBATM) is one which is restricted to use at most $n+1$ tape cells, where n is the length of the input. LBATM ACCEPTANCE is the problem of telling whether, given a LBATM M and string $s \in \Sigma^*$, $(s, q_0, 1)$ is an accepting configuration. LBATM ACCEPTANCE has been proven to be exponential time complete.

We make a polynomial reduction from LBATM ACCEPTANCE to our problem to show that it is EXPTIME-hard. Suppose we are given an LBATM M, and an input string $x = (x_1, x_2, \ldots, x_{n-1})$ such that $x_i \in \Sigma$ for each i. We can map this into the following planning problem $P(M, x)$:

Constant symbols: $\Gamma \cup K \cup \{p_1, p_2, \ldots, p_n\}$, where the p_i's are any n distinct symbols used to represent the position of the head.

Variable symbols: $\{v_1, v_2, \ldots, v_{n+2}\}$

Predicates: $\text{accept}(v_1, \ldots, v_{n+2})$. The first n arguments are used to store the contents of the tape, and the next two arguments are used to store the state and head position.

Operators: Let q be any state and A be any tape symbol.

If $\delta(q, A)$ contains (q', A', L) and q is an existential state, then there are operators $\{L_i^{q,A} : i = 2, \ldots, n\}$ as follows:

Name:	$L_i^{q,A}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$
Pre:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A', v_{i+1}, \ldots, v_n, q', p_{i-1})\}$
Del:	$\emptyset$
Add:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A, v_{i+1}, \ldots, v_n, q, p_i)\}$

If $\delta(q, A)$ contains (q', A', R) and q is an existential state, then there are operators $\{R_i^{q,A} : i = 1, \ldots, n-1\}$ as follows:

Name:	$R_i^{q,A}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$
Pre:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A', v_{i+1}, \ldots, v_n, q', p_{i+1})\}$
Del:	$\emptyset$
Add:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A, v_{i+1}, \ldots, v_n, q, p_i)\}$

If $\delta(q, A)$ contains (q', A', S) and q is an existential state, then there are operators $\{S_i^{q,A} : i = 1, \ldots, n\}$ as follows:

Name:	$S_i^{q,A}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$
Pre:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A', v_{i+1}, \ldots, v_n, q', p_i)\}$
Del:	$\emptyset$
Add:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A, v_{i+1}, \ldots, v_n, q, p_i)\}$

If q is a universal state, then there are operators $\{U_i^{q,A} : i = 1, \ldots, n\}$ as follows:

Name:	$U_i^{q,A}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$
Pre:	$\{\text{Pre}_L \cup \text{Pre}_R \cup \text{Pre}_S\}$
Del:	$\emptyset$
Add:	$\{\text{accept}(v_1, \ldots, v_{i-1}, A, v_{i+1}, \ldots, v_n, q, p_i)\}$

where

$$\begin{aligned}
\text{Pre}_L &= \{\text{accept}(v_1, \ldots, v_{i-1}, A', v_{i+1}, \ldots, v_n, q', p_{i-1}) | (q', A', L) \in \delta(q, A)\} \\
\text{Pre}_R &= \{\text{accept}(v_1, \ldots, v_{i-1}, A', v_{i+1}, \ldots, v_n, q', p_{i+1}) | (q', A', R) \in \delta(q, A)\} \\
\text{Pre}_S &= \{\text{accept}(v_1, \ldots, v_{i-1}, A', v_{i+1}, \ldots, v_n, q', p_i) | (q', A', S) \in \delta(q, A)\}
\end{aligned}$$

Note that Pre_L is empty if $i = 1$, and Pre_R is empty if $i = n$. Furthermore, the precondition list for this operator will be empty when δ does not contain any transitions for the state q. This is in accordance with the definition of an accepting configuration.

Initial state: The initial state is empty.

Goal condition: $\text{accept}(x_1, x_2, \ldots, x_{n-1}, \#, q_0, p_1)$.

The operators in the planning problem directly mimic the definition of an accepting configuration. Thus M accepts x if and only if there is a plan that achieves

$$\text{accept}(x_1, x_2, \ldots, x_{n-1}, \#, q_0, p_1).$$

Furthermore, $P(M, x)$ can be produced in low-order polynomial time. Thus, planning with no delete lists, no negation and no function symbols is EXPTIME-hard. ■

Theorem 5.10 If we restrict **P** to be context-free, positive, and deletion-free, then PLAN EXISTENCE is PSPACE-complete.

Proof. Below, we show that the problem is PSPACE-hard and that it is in PSPACE.

Hardness. This is established by showing that the acceptability problem for linearly bounded automatons (LBAs), which is known to be PSPACE-complete (Garey and Johnson [16]) reduces to the problem in polynomial time.

An LBA is normally denoted by $M = (K, \Sigma, \Gamma, \delta, q_0, F)$. $K = \{q_0, \ldots, q_m\}$ is a finite set of states. $F \subseteq K$ is the set of final states. Γ is the finite set of allowable tape symbols. $\Sigma \subseteq \Gamma$ is the set of allowable input symbols. $q_0 \in K$ is the start state. δ, the next move function, is a mapping from $K \times \Gamma$ to subsets of $K \times \Gamma \times \{\text{Left}, \text{Right}\}$. An LBA uses only n tape cells, where n is the length of the input string.

Suppose we are given an LBA M, and an input string $x = (x_1, x_2, \ldots, x_n)$ such that $x_i \in \Sigma$ for each i. We can map this into the following planning problem :

Constant symbols: $\Gamma \cup K \cup \{p_1, p_2, \ldots, p_n\}$, where the p_i's are any n distinct symbols used to represent the position of the head.

Variable symbols: $\{V_1, V_2, \ldots, V_{n+2}\}$

Predicates: configuration$(V_1, \ldots, V_{n+2})$, done(). The first n positions in configuration() are used to store the contents of the tape, and the next two positions are used to store the state and head position.

Operators:

Whenever $\delta(q, a)$ contains (q', a', Left), we create the following operator for each $p_i, i \neq 1$:

Pre: $\{\text{configuration}(V_1, \ldots, V_{i-1}, a, V_{i+1}, \ldots, V_n, q, p_i)\}$
Del: $\emptyset$
Add: $\{\text{configuration}(V_1, \ldots, V_{i-1}, a', V_{i+1}, \ldots, V_n, q', p_{i-1})\}$

Whenever $\delta(q, a)$ contains (q', a', Right), we create the following operator for each $p_i, i \neq n$:

Pre: $\{\text{configuration}(V_1, \ldots, V_{i-1}, a, V_{i+1}, \ldots, V_n, q, p_i)\}$
Del: $\emptyset$
Add: $\{\text{configuration}(V_1, \ldots, V_{i-1}, a', V_{i+1}, \ldots, V_n, q', p_{i+1})\}$

For each state $q \in F$ we create the following rule:

Pre: $\{\text{configuration}(V_1, \ldots, V_n, q, V_{n+2})\}$
Del: $\emptyset$
Add: $\{\text{done}()\}$

Initial state: configuration$(x_1, \ldots, x_n, q_0, p_1)$

Goal: done()

The operators directly mimic the possible actions of M on x. Thus M accepts x if and only if there is a plan for done. The transformation is obviously polynomial. Hence, the problem at hand is PSPACE-hard.

Membership. We present an algorithm below that demonstrates membership of our problem in NPSPACE. As NPSPACE = PSPACE, this establishes that the problem is in PSPACE. Intuitively, Q is the set of subgoals that have not been achieved yet.

1. $Q := G - S_0$.
2. If $Q = \emptyset$, then halt and accept.
3. Nondeterministically choose an operator instance α such that $Q \cap \text{Add}(\alpha) \neq \emptyset$.
4. $Q := (Q - \text{Add}(\alpha)) \cup (\text{Pre}(\alpha) - S_0)$.
5. Go to Step 2.

Note that size of Q never grows, as each operator has at most one precondition, hence it can be represented in PSPACE.

If there is a plan for the goal, then there will be a sequence of choices that would end up achieving all the subgoals, hence the algorithm would halt and accept; If there is no plan for the goal, no sequence of choices would end up achieving all the subgoals , and input will be rejected.

Since PSPACE = NPSPACE, we are done. ■

Theorem 5.11 If we restrict **P** to be deletion-free, positive, and context-free, then PLAN LENGTH is PSPACE-complete.

Proof. Below, we show that the problem is PSPACE and that it is PSPACE-hard.

Membership. We had proved the existence version of this problem to be in NPSPACE. All we need to do is to modify the previous algorithm so that we also verify the length of the plan found to be at most k. Since PSPACE = NPSPACE, this proves membership in PSPACE. Here is the algorithm:

1. $Q := G - S_0$; counter := k
2. If $Q = \emptyset$, then halt and accept.
3. Nondeterministically choose an operator instance α such that $Q \cap \mathrm{Add}(\alpha) \neq \emptyset$.
4. Decrement counter. Halt and reject if counter=0.
5. $Q := (Q - \mathrm{Add}(\alpha)) \cup (\mathrm{Pre}(\alpha) - S_0)$.
6. Go to step 2.

Hardness. We had proved the plan existence version of this problem to be PSPACE-hard. (Theorem 5.10) Since we do not have delete lists, the length of any plan need not exceed number of operator instances. We can reduce the existence version to this problem by setting k to this value.

■

Theorem 5.12 PLAN LENGTH is NEXPTIME-complete in each of the following cases:

1. **P** is deletion-free and positive;
2. **P** is deletion-free;
3. **P** is positive;
4. no restrictions (except, of course, that **P** is function-free).

Proof.

Membership. Since k is part of the input, and it is encoded in binary, k can be at most exponential in terms of length of the input. We can nondeterministically guess a sequence of instances of the operators, and in exponential time, we can verify that it is a plan of length at most k that achieves the goal. Hence the problems are in NEXPTIME.

Hardness. Next, we show that Case 1, which is a special case of Cases 2, 3, and 4, is NEXPTIME-hard. Given a nondeterministic Turing machine M, that runs in exponential time, and an input string $x = x_0, \ldots, x_{n-1}$, we reduce it to the optimum planning problem without delete lists and negated preconditions

Without loss of generality we assume when the TM enters a halting state, $\delta()$ will be such that it will stay in the same state, write the same symbol it reads, and the head position will remain stationary.

We create the following planning problem:

Predicates: greater($\underline{i}, \underline{j}$) is a $2n$-ary predicate to assert that i is greater than j. It is used to assert instances of the "diff" predicate.

diff($\underline{i}, \underline{j}$) is a $2n$-ary predicate used to assert that i and j are different numbers.

choicecounter($\ldots$) is an n-ary predicate used when making the nondeterministic choices.

choice($\underline{i}, p$) is an $n+1$-ary predicate. The first n places encode the step in binary, the $n+1$th place holds the nondeterministic choice for this step.

filltape($\ldots$) is an n-ary predicate used as a counter while initializing the blank portion of the tape.

counter($\underline{i}, \underline{j}$) is a $2n$-ary predicate denoting that the j'th tape cell at step i is being processed.

contains($\underline{i}, \underline{j}, c$) is a $2n+1$-ary predicate denoting that at i'th step tape cell j contains c.

state($\underline{i}, q$) is a $n+1$-ary predicate holding the current state at step i.

$h(\underline{i}, \underline{j})$ is a $2n$-ary predicate denoting that at step i, the head is at position j.

laststate(q) is a unary predicate denoting the state after the last (2^n) step.

done() is a propositional symbol denoting that the TM accepted the input string.

Initial state: $\{\text{diff}(\underline{0}, \underline{1}), \text{diff}(\underline{1}, \underline{0}), \text{greater}(\underline{1}, \underline{0})\} \cup \{\text{contains}(\underline{0}, \underline{i}, x_i) : i < n\}$

Goal: done()

Operators: Each operator below that contains increment or decrement operations (such as mapping i to $i+1$ or $i-1$) should be expanded as described in Section B.1.

The following operators assert instances of the "diff" predicate.

Name:	$D(\underline{i}, \underline{j})$
Pre:	$\{\text{greater}(\underline{i}, \underline{j})\}$
Add:	$\{\text{greater}(\underline{i+1}, \underline{j}), \text{diff}(\underline{i+1}, \underline{j}), \text{diff}(\underline{j}, \underline{i+1})\}$
Del:	$\emptyset$

Name:	$G(\underline{j})$
Pre:	$\{\text{greater}(\underline{2^n-1}, \underline{j})\}$
Add:	$\{\text{greater}(\underline{j+2}, \underline{j+1}), \text{diff}(\underline{j+2}, \underline{j+1}), \text{diff}(\underline{j+1}, \underline{j+2})\}$
Del:	$\emptyset$

Pre:	$\{\text{greater}(\underline{2^n-1}, \underline{2^n-2})\}$
Add:	$\{\text{choicecounter}(\underline{0})\}$
Del:	$\emptyset$

The following operator makes the nondeterministic choices of the TM for every step:

Name:	$C(\underline{i}, V)$
Pre:	$\{\text{choicecounter}(\underline{i})\}$
Add:	$\{\text{choicecounter}(\underline{i+1}), \text{choice}(\underline{i}, V)\}$
Del:	$\emptyset$

Name:	$Cl(V)$
Pre:	$\{\text{choicecounter}(\underline{2^n-1})\}$
Add:	$\{\text{choice}(\underline{2^n-1}, V), \text{filltape}(\underline{n})\}$
Del:	$\emptyset$

The following operators initialize the blank portion of the tape:

Name: $F(\underline{i})$
Pre: $\{\text{filltape}(\underline{i})\}$
Add: $\{\text{filltape}(\underline{i+1}), \text{contains}(\underline{0}, \underline{i}, \#)\}$
Del: $\emptyset$

Pre: $\{\text{filltape}(\underline{2^n-1})\}$
Add: $\{\text{counter}(\underline{0}, \underline{0}), \text{contains}(\underline{0}, \underline{2^n-1}, \#)\}$
Del: $\emptyset$

The following two operators copy the contents of the tape at step i to step $i+1$:

Name: $\text{Copy}(\underline{i}, \underline{j}, \underline{K}, V)$
Pre: $\{\text{counter}(\underline{i}, \underline{j}), h(\underline{i}, \underline{K}), \text{diff}(\underline{j}, \underline{K}), \text{contains}(\underline{i}, \underline{j}, V)\}$
Add: $\{\text{counter}(\underline{i}, \underline{j+1}), \text{contains}(\underline{i+1}, \underline{j}, V)\}$
Del: $\emptyset$

Name: $\text{Copyl}(\underline{i}, \underline{k}, V)$
Pre: $\{\text{counter}(\underline{i}, \underline{2^n-1}), h(\underline{i}, \underline{k}), \text{diff}(\underline{i}, \underline{k}), \text{contains}(\underline{i}, \underline{2^n-1}, V)\}$
Add: $\{\text{contains}(\underline{i+1}, \underline{2^n-1}, V), \text{counter}(\underline{i+1}, \underline{0})\}$
Del: $\emptyset$

The following imitates the moves of the Turing machine, by writing the appropriate symbol on the current cell, changing the state, and moving the head. Whenever (q', c', t) is the p'th element in $\delta(q, c)$ we have the operators:

Name: $M_{q,c,p}^{q',c',t}(\underline{i}, \underline{j})$
Pre: $\{\text{counter}(\underline{i}, \underline{j}), h(\underline{i}, \underline{j}), \text{state}(\underline{i}, q), \text{contains}(\underline{i}, \underline{j}, c), \text{choice}(\underline{i}, p)\}$
Add: $\{\text{counter}(\underline{i}, \underline{j+1}), \text{state}(\underline{i+1}, q'), \text{contains}(\underline{i+1}, \underline{j}, c') h(\underline{i+1}, \underline{j+d})\}$

d is 1 if t is right, -1 if t is left, and 0 otherwise.

Name: $\text{Ml}_{q,c,p}^{q'}(\underline{i}, \underline{j})$
Pre: $\{\text{counter}(\underline{2^n-1}, \underline{0}), h(\underline{2^n-1}, \underline{j}), \text{state}(\underline{2^n-1}, q),$
$\text{contains}(\underline{2^n-1}, \underline{j}, c), \text{choice}(\underline{i}, p)\}$
Add: $\{\text{laststate}(q')\}$

For each $q \in F$ we have the following operator:

Name: $F_q()$
Pre: $\{\text{laststate}(q)\}$
Add: $\{\text{done}()\}$

The planning system works in phases. In the first phase, instances of the "diff" predicate are asserted. In the end of this phase the next phase which makes the nondeterministic choices is enabled. In the end of this , the next phase, which initializes the blank portion of the tape is enabled. When the tape is filled with blanks, we enable the next phase, which actually mimics the behavior of the TM. We make use of the predicate counter(i, j) in this phase. Suppose, we are at step i and we are examining cell j. if head is not in position j at that step, we simply copy the contents of this cell to the next step. If head is at position j at that step, we make the move according to the nondeterministic choice that we have made before, and set the new state, head position, contents of cell j for the next step. Then we consider the cell $j + 1$. When we reach the end of the tape (cell $2^n - 1$), we turn back to cell 0 and proceed with the next step. In the very last step (step 2^n) we do not need to do all this. We just determine what the next state would be. Then if this state is a final state, we assert "done()."

Note that each operator enables the next one, hence there is no plan that would follow a different order. The only remaining problem is ensuring that the operator that makes the nondeterministic choices does not fire more than once for the same step. We do this by putting a bound on the length of the plan so that if we make more than one nondeterministic choice at some step, the remaining number of steps will not be enough to complete the plan.

We spend $(2^n - 1)2^{n-1}$ steps in asserting instances of the "diff" predicate, 2^n steps in making the nondeterministic choices, $2^n - n$ steps for initializing the tape, $(2^n - 1)2^n$ steps for simulating the moves, 1 step for the last move, and 1 step for asserting done(), giving $k = 3 \times 2^{2n-1} + 2^{n-1} - n + 2$ in total.

The Turing machine accepts x iff there exists a plan of length k that achieves done. The reduction is obviously polynomial. ■

B.4 Planning When the Operator Set is Fixed

Theorem 5.13 PLAN EXISTENCE can be solved in constant time if we restrict $\mathbf{P} = (S_0, \mathcal{O}, G)$ to be propositional and $\mathcal{O}$ to be a fixed set.

Proof. Both the number of operators, and the number of propositions we need to consider are constant, which implies that the number of possible plans and their lengths are bounded by a constant. Thus we can solve the planning problem in constant time. ■

Corollary 5.5 PLAN LENGTH can be solved in constant time if we restrict $\mathbf{P} = (S_0, \mathcal{O}, G)$ to be propositional and $\mathcal{O}$ to be a fixed set.

Proof. Since the number of possible plans is constant, we can check all of them in constant time. ■

Theorem 5.14

1. If we restrict $\mathbf{P}$ to be fixed, deletion-free, context-free and positive, then PLAN EXISTENCE is in NLOGSPACE and PLAN LENGTH is in NP.

2. If we restrict $\mathbf{P}$ to be fixed, deletion-free, and positive, then PLAN EXISTENCE is in P and PLAN LENGTH is in NP.

3. If we restrict $\mathbf{P}$ to be fixed and deletion-free, then PLAN EXISTENCE and PLAN LENGTH are in NP.

4. If we restrict $\mathbf{P}$ to be fixed, then PLAN EXISTENCE and PLAN LENGTH are in PSPACE.

Proof. When the set of operators is fixed, we can enumerate all ground instances in polynomial time, reducing the problem to propositional planning. Hence the theorem follows from propositional planning results. ■

Theorem 5.15 There exists a fixed positive deletion-free set of operators $\mathcal{O}$ for which PLAN LENGTH is NP-hard.

Proof. Here are the operators:

Pre: $\{\text{counter}(X), \text{next}(X,Y)\}$
Add: $\{\text{counter}(Y), \text{true}(X)\}$
Del: $\emptyset$

Pre: $\{\text{counter}(X), \text{next}(X,Y)\}$
Add: $\{\text{counter}(Y), \text{false}(X)\}$
Del: $\emptyset$

Pre: $\{\text{counter}(\text{last}), \text{poslit}(X,C), \text{true}(X)\}$
Add: $\{\text{done}(C)\}$
Del: $\emptyset$

Pre: $\{\text{counter}(\text{last}), \text{neglit}(X,C), \text{false}(X)\}$
Add: $\{\text{done}(C)\}$
Del: $\emptyset$

We can reduce the satisfiability problem, which is known to be NP-complete, to this problem as follows.

Given a boolean expression E in CNF form containing variables $\{x_1, \ldots, x_n\}$, we output the following:

$$\begin{aligned} k &= n + \text{the number of clauses in } E \\ G &= \{\text{done}(c) : c \text{ is a clause of } E\} \\ S_0 &= \{\text{poslit}(x,c) : x \text{ is an atom of } c\} \cup \{\text{neglit}(x,c) : x \text{ is a negative literal of } c\} \\ &\quad \cup \{\text{next}(x_i, x_{i+1}) : i = 1, \ldots, n-1\} \cup \{\text{next}(n, \text{last})\} \end{aligned}$$

The "counter" predicate is used to ensure that the operators that assign truth values to variables of the boolean expression, are enabled sequentially. Together with the bound on the plan length, this ensures that each variable is assigned a unique truth value. E is satisfiable iff there exists a plan of length k. The reduction is clearly polynomial. ■

Theorem 5.16 There exist fixed deletion-free sets of operators $\mathcal{O}$ for which PLAN EXISTENCE and PLAN LENGTH are NP-hard.

Proof. NP-hardness for PLAN LENGTH follows from Theorem 5.15. Here are the operators for which PLAN EXISTENCE is NP-hard:

Pre: $\{\neg\text{true}(X)\}$
Add: $\{\text{false}(X)\}$
Del: $\emptyset$

Pre: $\{\neg\text{false}(X)\}$
Add: $\{\text{true}(X)\}$
Del: $\emptyset$

Pre: $\{\text{poslit}(X, C), \text{true}(X)\}$
Add: $\{\text{done}(C)\}$
Del: $\emptyset$

Pre: $\{\text{neglit}(X, C), \text{false}(X)\}$
Add: $\{\text{done}(C)\}$
Del: $\emptyset$

Intuitively, false(X) and true(X) stands for X is asserted true and false respectively. poslit(X,C) and neglit(X,C) stands for the assertions that X is a positive literal of clause C, and X is a negative literal of clause C, respectively.

We reduce the satisfiability problem (which is known to be NP-complete) to this problem. Given a boolean expression in CNF form, we create the initial state as

$$\begin{aligned} S_0 &= \{\text{poslit}(x, c) : x \text{ is an atom of } c\} \cup \{\text{neglit}(x, c) : x \text{ is a negative literal of } c\} \\ G &= \{\text{done}(c) : c \text{ is a clause}\} \end{aligned}$$

The expression is satisfiable iff there exists a plan to assert done(c) for each clause c. The reduction is clearly polynomial. ■

Theorem 5.17 There exists a fixed set of positive operators $\mathcal{O}$ for which PLAN EXISTENCE and PLAN LENGTH are PSPACE-hard.

Proof. Here is the set of operators:

Name: $R(I, J, V, Q, S, Y)$
Pre: $\{\text{head}(I), \text{next}(I, J), \text{contains}(I, V), \text{state}(Q), \text{delta}(Q, V, S, Y, \text{Right})\}$
Add: $\{\text{head}(J), \text{contains}(I, Y), \text{state}(S)\}$
Del: $\{\text{head}(I), \text{contains}(I, V), \text{state}(Q)\}$

Name: $L(I, J, V, Q, S, Y)$
Pre: $\{\text{head}(I), \text{next}(J, I), \text{contains}(I, V), \text{state}(Q), \text{delta}(Q, V, S, Y, \text{Left})\}$
Add: $\{\text{head}(J), \text{contains}(I, Y), \text{state}(S)\}$
Del: $\{\text{head}(I), \text{contains}(I, V), \text{state}(Q)\}$

Name: $D(Q)$
Pre: $\{\text{state}(Q), \text{final}(Q)\}$
Add: $\{\text{done}()\}$
Del: $\emptyset$

We can reduce the linearly bounded automata (LBA) acceptance to this problem as follows. Given a TM M that is linearly bounded, and an input string $x = x_1, \ldots, x_n$ we create the initial state and the goal

$$\begin{aligned} S_0 &= \{\text{next}(p_i, p_{i+1}) : i = 1, \ldots, n-1\} \cup \{\text{contains}(p_i, x_i) : i = 1, \ldots, n\} \\ &\quad \cup \{\text{delta}(Q, V, S, Y, \text{Left}) : (S, Y, \text{Left}) \in \delta(Q, V)\} \\ &\quad \cup \{\text{delta}(Q, V, S, Y, \text{Right}) : (S, Y, \text{Right}) \in \delta(Q, V)\} \\ &\quad \cup \{\text{final}(Q) \ : Q \in F\} \cup \{\text{state}(q_0), \text{head}(p_1)\} \\ G &= \{\text{done}()\} \end{aligned}$$

The operators mimic the moves of the LBA, one for one. The LBA accepts x iff there is a plan that achieves done(). The reduction is obviously polynomial. Thus PLAN EXISTENCE with this set of operators is PSPACE-hard.

The same set of operators works for PLAN LENGTH well. Since the number of distinct LBA configurations is exponential in terms of the input string length, LBA halts within that many moves. Since the planning operators mimic the LBA move for move, all we need to do is set $k =$ the number of configurations. Then PLAN EXISTENCE is just a special case of it. ∎

Theorem 5.18 (Complexity of planning with conditional operators) Theorems 5.3 through 5.5, 5.7 through 5.17, and their corollaries still hold when $\mathcal{O}$ is allowed to contain conditional operators.

Proof.

Varying set of operators. Hardness follows directly because planning with regular operators is a special case of planning with conditional operators, where each operator is restricted to contain exactly one triple. For membership, we will outline the modifications needed for each case. The modifications for both propositional planning and datalog planning will be the same, so we will not state them twice.

PLAN EXISTENCE:

- *Deletion-free, positive, context-free.* The same algorithm given for membership will work, except when we choose an operator, we should nondeterministically choose a subset of the triples such that each triple chosen achieves at least one distinct subgoal, and use the add list and preconditions of all these triples.

- *Deletion-free, positive.* The membership proof is based on the fact that no operator needs to be applied more then once (because it is deletion-free), and the order of the operators does not matter (because they are positive). When we have conditional operators, the only difference is that each operator needs to be used at most n times, where n is the number of triples it contains. This difference does not affect the proof at all.

- *Deletion-free.* As mentioned above, we do not need to use any operator more times than the number of triples it contains. Hence the lengths of the plans are still at the same level (exponential for datalog, polynomial for propositional case) as before. Thus, the same proof as before works.

- *Unrestricted.* The algorithm given for this proof was based on a non-deterministic forward search. Since state size is not affected by conditional operators, the same algorithm still works. The only difference is, when we have chosen the operator, and computing the next state, we need to use the effects of all the triples whose precondition lists are satisfied in the current state.

PLAN LENGTH:

- *Datalog operators.* The same NEXPTIME proof will work, since k still confines us to plans of exponential length.

- *Context-free,positive, deletion-free.* The same algorithm used for PLAN EXISTENCE will work, if we introduce a counter, initialize it to k, and decrement it at each iteration. We add a new step that checks the counter, and fails when it hits 0.
- *Propositional planning (unrestricted).* Same modification as the previous one.
- *Propositional, deletion-free.* since $\mathbf{P}$ is deletion-free, each operator instance need to be used no more than the number of triples it contains. Thus the plans need not be longer than polynomial, and nondeterministically, we can choose a sequence of operator instances and verify that it is a plan that achieves the goal, and that it is at most of length k.

Fixed set of operators. Propositional planning with a fixed set of operators is obviously still constant time. As before, the results for datalog planning follow from the results about propositional planning with a varying set of operators. ■