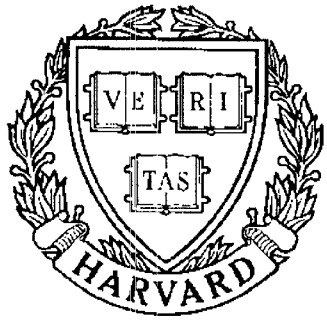
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

Complexity, Decidability and Undecidability Results for Domain-Independent Planning

by K. Erol, D.S. Nau, and V.S. Subrahmanian

TR 91-96

Complexity, Decidability and Undecidability Results for Domain-Independent Planning*

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-like 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 [4], 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.

*This work was supported in part by NSF Grant NSFD CDR-88003012 to the University of Maryland Systems Research Center, as well as NSF grants IRI-8907890 and IRI-9109755.

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

‡Department of Computer Science, Systems Research Center, 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: Complexity of domain-independent planning.

Lang. restrictions	How the operators are given	Delete lists are allowed	Negated preconditions are allowed	Telling whether a plan exists	Telling whether there is a plan of length $\leq k$
none	either way	yes/no	yes/no/no$^{\alpha}$	semidecidable	semidecidable
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$^{\alpha}$	PSPACE-comp.	PSPACE-comp.
	fixed (in advance)	yes	yes/no	PSPACE $^{\beta}$	PSPACE $^{\beta}$
		no	yes	NP $^{\beta}$	NP $^{\beta}$
			no	P	NP $^{\beta}$
			no$^{\alpha}$	NLOGSPACE	NP
all predicates are 0-ary (propositions)	given in the input	yes	yes/no	PSPACE-comp. $^{\gamma}$	PSPACE-comp.
		no	yes	NP-complete $^{\gamma}$	NP-complete
			no	P $^{\gamma}$	NP-complete
			no$^{\alpha}$/no$^{\delta}$	NLOGSPACE-comp.	NP-complete
	fixed	yes/no	yes/no	constant time	constant time

$^{\alpha}$No operator has more than one precondition.

$^{\beta}$With PSPACE- or NP-completeness for some sets of operators.

$^{\gamma}$Results due to Bylander [2].

$^{\delta}$Every operator with more than one precondition is the composition of other operators.

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 [13, 3, 12, 20] and task reduction [15, 3, 20] 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 [4, 2, 7, 8, 10, 11]. This research has yielded some surprising results. For example, Gupta and Nau [7, 8] have shown that contrary to prior expectations, deleted-condition interactions are easy to handle in blocks-world planning.

In the current paper, we examine 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. If function symbols are allowed or if the language contains infinitely many constant symbols, then determining whether a plan exists is *undecidable* (more specifically,

semidecidable). This is true even if we have no delete lists and the precondition list of each operator contains at most one (non-negated) atom.

2. 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. In this case, the computational complexity varies from constant time to EXPSPACE-complete, depending on 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), and whether we fix the planning operators in advance or give them as part of the input. The results are summarized in Table 1.

3. The above results resolve an open problem stated by Chapman in [4]: whether or not planning is undecidable if the initial state is finite. If the initial state is finite and the language has finitely many ground terms, then planning is decidable, but if the initial state is finite and the language has infinitely many ground terms, then planning is undecidable.

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 situation" [4]. Our results show that this theorem, as stated, is misleading. Whether or not planning is undecidable has nothing to do with whether or not the operators have conditional effects—instead, planning is decidable if and only if the language contains finitely many ground terms.

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 contains concluding remarks. The proofs of the theorems and lemmas appear in the appendix.

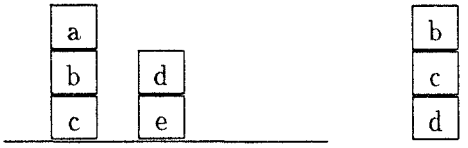
2 Preliminaries

In this section, we set out the precise definition of a planning domain. Throughout this section, we let $\mathcal{L}$ be an ordinary first-order language generated by finitely many constant, function and predicate symbols. $\mathcal{L}$ has an infinite number of variable symbols together with the usual logical symbols.

Definition 2.1 A *state* is a set of ground atoms.

Intuitively, a state tells us which ground atoms are currently true. Thus, if a ground atom A is in state S, then A is true in state S; 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 A *planning operator* α is a 4-tuple $(\mathrm{Name}(\alpha), \mathrm{Pre}(\alpha), \mathrm{Add}(\alpha), \mathrm{Del}(\alpha))$, where



(a) initial configuration (b) goal configuration

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

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

Observe that atoms and negated atoms may occur in the precondition list, but negated atoms may not occur in either the add list or the delete list.

When defining a planning operator α, often Name(α) will be clear from context. In such cases, we will not always specify Name(α) 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.

Definition 2.4 A *goal* is an existentially closed conjunction of atoms.

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 [12]. Suppose the initial configuration is as shown in Fig. 1(a), and the goal is to have b on c and 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 predicate symbol of arity 0), "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:

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

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 $X_i, 1 \leq i \leq n$. Suppose that the following conditions hold:

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

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

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

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 $S_0, \ldots, S_n, S_{n+1}$ of states, a sequence $\alpha_0, \ldots, \alpha_n$ of planning operators, and a sequence $\theta_0, \ldots, \theta_n$ of substitutions such that

$$S_0 \stackrel{\alpha_0,\theta_0}{\Longrightarrow} S_1 \stackrel{\alpha_1,\theta_1}{\Longrightarrow} S_2 \cdots \stackrel{\alpha_n,\theta_n}{\Longrightarrow} S_{n+1} \qquad (1)$$

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. $\mathcal{O}$ (and thus $\mathbf{P}$) is *positive* if for all $\alpha \in \mathcal{O}$, $\text{Pre}(\alpha)$ is a finite set of *atoms* (i.e. negations are not present in $\text{Pre}(\alpha)$).

2. $\mathcal{O}$ (and thus $\mathbf{P}$) is *deletion-free* if for all $\alpha \in \mathcal{O}$, $\text{Del}(\alpha) = \emptyset$.

3. $\mathcal{O}$ (and thus $\mathbf{P}$) is *context-free* if for all $\alpha \in \mathcal{O}$, $|\text{Pre}(\alpha)| \leq 1$, i.e., $\text{Pre}(\alpha)$ contains at most one atom.

4. $\mathcal{O}$ (and thus $\mathbf{P}$) is *side-effect-free* if for all $\alpha \in \mathcal{O}$, $|\text{Add}(\alpha) \cup \text{Del}(\alpha)| \leq 1$, i.e., α has at most one postcondition.

5. $\mathcal{L}$ (and thus $\mathbf{P}$) is *function-free* if $\mathcal{L}$ contains no function symbols.

6. $\mathcal{L}$ (and thus $\mathbf{P}$) is *propositional* if every predicate P in $\mathcal{L}$ is a propositional symbol (i.e. a predicate symbol of arity 0).

Note that if $\mathbf{P}$ is propositional then in effect it is also function-free, for even if $\mathcal{L}$ contains function symbols, no operator will ever use them.

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:

> 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.[1] As a consequence of this equivalence, we can use results on the complexity of logic programs and deductive databases to demonstrate the following results:

- PLAN EXISTENCE is *undecidable* even if:
  - we have no delete lists
  - we have no delete lists and preconditions of all operators contain at most one atom

  The existence or non-existence of negations in preconditions of planning operators makes no difference when decidability is concerned (though, as we shall see later, it does make a difference when we study the complexity of decidable planning domains).

[1] 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 [17]) in this paper. It doesn't make much sense to consider non-Herbrand models for planning problems because the domains of such models often contain objects that do not occur in the language. 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 [14, 21].

- 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 constants are present in the language). The presence or absence of delete lists does not affect the decidability result.

- 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 $\mathbf{P}$. The presence or absence of delete lists does not affect this result.

We now proceed to establish the equivalence between logic programming and planning without delete lists. Subsequently, 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

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

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 when $\mathbf{P} = (S_0, \mathcal{O})$ contains no operators that have negations in their preconditions. This irrelevance is due to the compactness theorem for first-order logic.

Lemma 3.1 *Suppose that* $\mathbf{P} = (S_0, \mathcal{O})$ *is any* positive, deletion-free *planning domain, and*

$$S_0 \stackrel{\alpha_0,\theta_0}{\Longrightarrow} S_1 \stackrel{\alpha_1,\theta_1}{\Longrightarrow} S_2 \cdots \stackrel{\alpha_n,\theta_n}{\Longrightarrow} S_{n+1}$$

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

For proofs of this and other results, see the appendix.

Note that if $\mathbf{P} = (S_0, \mathcal{O})$ is a positive deletion-free planning domain, then $\mathrm{LP}(\mathbf{P})$ is a *pure* (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 $\mathrm{LP}(\mathbf{P})$.

Theorem 3.1 (Equivalence Theorem) *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 $\mathrm{LP}(\mathbf{P}) \models G$.*

Corollary 3.1 (Semi-Decidability Results)

1. *$\{G \mid G$ 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 $\{A \mid A$ 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.*

The following result is an immediate corollary of Theorem 3.1.

Corollary 3.4 *Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a fixed 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* LP($\mathbf{P}$) *is decidable.*

Theorem 3.1 holds only when $\mathbf{P}$ is positive. The reason for this is that if $\mathbf{P}$ is not positive, then LP($\mathbf{P}$) is a logic program that may contain negation in its body. Logic programming interprets negation in 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 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 two operators α_1, α_2 described below:

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

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

$a \leftarrow \neg b$
$b \leftarrow c$
$c \leftarrow$

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.

Lemma 3.2 *Suppose I is a decidable Herbrand interpretation, i.e. I is a decidable set of ground atoms, and G is a goal. Then*

1. *The problem "is goal G true in interpretation I?" is semi-decidable.*
2. *If the language $\mathcal{L}$ is known to be function free, then the above problem is decidable.*

Theorem 3.2 (Decidability of Deletion-Free, Function-Free Planning) *If we restrict* $\mathbf{P}$ *to be deletion-free and function-free, then* PLAN EXISTENCE *is decidable.*

Theorem 3.3 (Theorem on Infinite Initial States) PLAN EXISTENCE *is semi-decidable if we restrict* $\mathbf{P} = (S_0, \mathcal{O})$ *to be positive and deletion-free, and* S_0 *to be a* decidable *set of ground atoms of language* $\mathcal{L}$ *(even though* S_0 *may be infinite).*

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 [6, 12] (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) *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

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

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 $\mathbf{P}$. 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 3.2 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.*

4 Comparison with Chapman's Undecidability Results

To date, the best-known results on decidability and undecidability in planning systems are those of Chapman [4]. 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 ([4, 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. Our results compare and contrast with this result in the following respects:

1. Chapman assumes that "an infinite set of constants t_i are used to represent the tape squares" [4, p. 371], but in his discussion of the result, he points out that this set of constants is recursive.

   Our language $\mathcal{L}$ contains only finitely many constants, but it may contain function symbols. This is essentially the same as having a recursive set of constants, because the function symbols can be used to generate countably many ground terms. Thus in this respect, our result doesn't differ from Chapman's first undecidability theorem.

2. Chapman uses an infinite initial state to prove his result. 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)" [4, p. 371]. Chapman also says [4, p. 344]:

   > 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.)

   We now describe how our results solve the open problem posed in the above quote.

   First, our Corollary 3.1 shows that if the language contains infinitely many ground terms, then planning is undecidable even if initial states are finite. This is true regardless of whether the infinite number of ground terms occurs because of infinitely many constants being present in the language as is the case with Chapman[4], or because there are finitely many constants together with finitely many function symbols, as is true in our case.

   Second, our result Theorem 3.5 shows that if the language contains only finitely many ground terms, then first-order planning is decidable. Planning with only finitely many ground terms in the language makes a lot of sense, because it applies to all domains where a finitely bounded number of "entities" are being manipulated (the

usual formulation of the blocks world is one such example). We can make the number of these entities large, but as long as we keep it finite, these results apply.

3. Our Corollaries 3.1 and 3.3 demonstrate that first-order planning with infinitely many terms is undecidable even if every planning operator contains no delete lists, at most one positive precondition, and no negative preconditions. Chapman's result [4, p. 371] is more restrictive: he needs four preconditions, and he explicitly uses negative post-conditions, which is equivalent to having delete lists.

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 situation" [4, p. 373]. Chapman does not state explicitly what he means by "actions whose effects are a function of their input situation", but he appears to mean something like the following kind of conditional operator:

if P_1 then add A_1 and delete D_1
else if P_2 then add A_2 and delete D_2
...
else if P_{n-1} then add A_{n-1} and delete D_{n-1}
else if P_n then add A_n and delete D_n

where for each i, P_i and A_i and D_i are sets of atoms. All of the operators used by Chapman in the proof of this theorem are special cases of this kind of conditional operator.

A careful reading of Chapman's proof makes it clear that there is an additional assumption. In his proof, he makes use of operators that increment and decrement two counters, but there is no upper bound on the value of those counters—and thus it is necessary that the language contain infinitely many terms. But if there are infinitely many terms, then our Corollary 3.1 shows that planning is undecidable even with ordinary first-order planning operators.

On the other hand, suppose that the language contains only finitely many terms, but that there are conditional operators such as those described above. Any such conditional operator is equivalent to the following set of first-order planning operators $\{O_i | i = 1, \ldots, n\}$:

Name :	$O_i(\ldots)$
Pre :	$\{\neg a \mid a \in P_1 \cup \ldots \cup P_{i-1}\} \cup P_i$
Add :	A_i
Del :	D_i

Replacing the conditional operator by the equivalent set of first-order operators yields a first-order planning domain whose language has finitely many terms—and according to Theorem 3.5, planning is decidable for such domains.

From the above, it follows that the statement of Chapman's Second Undecidability Theorem is misleading. His proof of undecidability has nothing to do with whether the operators are conditional—it instead depends on the fact that his planning domain requires an infinite number of terms.

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 present the effect of delete lists, negated preconditions, propositional operators, and fixing the set of operators on the complexity of planning.

5.1 Preliminaries for the Complexity Results

5.1.1 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 :	$\{c(i_1, i_2, \ldots, i_{n-2}, 1, 0)\}$

...

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:

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

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:

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

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

$$\begin{aligned} \text{Name}: &\quad \text{incr}(\underline{i}) \\ \text{Pre}: &\quad \{c(\underline{i})\} \\ \text{Del}: &\quad \{c(\underline{i})\} \\ \text{Add}: &\quad \{c(\underline{i+1})\} \end{aligned}$$

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 $c(i_1, i_2, \ldots, i_n)$.

5.1.2 Eliminating Negated Preconditions

In Theorem 3.4, we proved that delete lists could be "compiled away" in exponential time into a planning domain that introduced no new negations. 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, we show that negated preconditions do not affect the complexity of planning in the presence of delete lists.

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).*

Synopsis of proof. The basic idea is this:[2] 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

5.2.1 Propositional Planning

Theorem 5.2 (Bylander)

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, deletion-free, and positive 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*.*

For the proof, the reader is referred to Bylander [2].

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

[2] For complete proofs of this and the other results, see the appendix.

Synopsis of proof. We give an encoding of off-line, nondeterministic logspace Turing machine for the hardness proof. The proof of membership uses the fact that each operator has at most one precondition. Furthermore, since they are restricted to be positive and deletion-free, the subgoals do not interact. As a result we can follow a chain from the goal state to initial state. ■

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

Synopsis of proof. We reduce the Set Cover problem to this problem as follows. For each set element we define a proposition. For each subset in the cover set, we define an operator with empty delete list and empty precondition list. The add list contains the propositions in the subset. The goal is the collection of propositions corresponding to set members. Since **P** is restricted to be propositional and deletion-free, the length of the plans need not exceed number of propositions, which is polynomial in terms of the input. We can nondeterministically choose a sequence of operators (of size polynomial) and verify that it is a plan of length at most k that achieve the goal. Thus, membership follows. ■

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

Synopsis of proof. Hardness follows from Theorem 5.4. For membership, we use the fact that we can put a polynomial bound on the plan length, as **P** is propositional and deletion-free. ■

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

Synopsis of proof. Hardness follows from Theorem 5.4. For membership, we use the fact that we can put a polynomial bound on the plan length, as **P** is propositional and deletion-free. ■

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.*

Synopsis of proof. The proof is accomplished by defining reductions from the plan existence versions of these problems. The reduction is done simply by setting $k = 2^n$, where n is the length of the input. ■

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 $\text{del}(\alpha)$ are disjoint, and the negative preconditions of β and $\text{add}(\alpha)$ are disjoint.

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

$$\begin{aligned} \text{Pre}: &\quad \text{Pre}(\alpha) \cup (P_1 - \text{Add}(\alpha)) \cup (P_2 - del(\alpha)) \\ \text{Add}: &\quad \text{Add}(\beta) \cup (\text{Add}(\alpha) - \text{Del}(\beta)) \\ \text{Del}: &\quad \text{Del}(\beta) \cup (\text{Del}(\alpha) - \text{Add}(\beta)) \end{aligned}$$

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 [2] 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:

$$\begin{aligned} \text{Name}: &\quad \text{totable}_{ij} \\ \text{Pre}: &\quad \{\text{off}_{1,i}, \text{off}_{2,i}, \ldots, \text{off}_{n,i}, \text{off}_{1,j}, \text{off}_{2,j}, \ldots, \text{off}_{i-1,j}, \text{off}_{i+1,j}, \ldots, \text{off}_{n,j}\} \\ \text{Del}: &\quad \emptyset \\ \text{Add}: &\quad \{\text{off}_{i,j}\} \end{aligned}$$

$$\begin{aligned} \text{Name}: &\quad \text{toblock}_{ij} \\ \text{Pre}: &\quad \{\text{off}_{1,i}, \text{off}_{2,i}, \ldots, \text{off}_{n,i}, \text{off}_{1,j}, \text{off}_{2,j}, \ldots, \text{off}_{n,j}, \text{off}_{i,1}, \text{off}_{i,2}, \ldots, \text{off}_{i,n}\} \\ \text{Del}: &\quad \{\text{off}_{i,j}\} \\ \text{Add}: &\quad \emptyset \end{aligned}$$

In Bylander's formulation of blocks world, **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 [8]), 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 [3, 9, 12, 19, 22, 7, 8]. Thus, from results proved in [8], it follows that in this extension of Bylander's formulation, PLAN LENGTH is NP-complete.

5.2.2 Function-Free First-Order Planning

Theorem 5.7 *If we restrict* **P** *to be positive and deletion-free, then* PLAN EXISTENCE *is* EXPTIME-*complete.*

Synopsis of proof. We give a reduction from the linearly bounded alternating Turing machine (LBATM) acceptance problem, which is known to be EXPTIME-complete. The reader is referred to Stockmeyer and Chandra [18] for the proof. An LBATM uses at most n tape cells, where n is one plus the length of the input string. We make use of a predicate "accept" having arity $n + 2$. The first n positions represent the contents of the tape, the next two positions represent the current state, and head position. Intuitively, accept(...) is true of a machine configuration iff it is an accepting configuration of the LBATM. The operators mimic the definition of an accepting configuration. ■

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

Synopsis of proof. We define a reduction from NEXPTIME Turing machine acceptance problem. We use atoms contains($\underline{i}, \underline{j}, c$) state($\underline{i}, q$), head($\underline{i}, \underline{t}$) to denote that at the $\underline{i}^{th}$ step, the $\underline{j}^{th}$ cell contains c, state is q, and head is at position $\underline{t}$. Negated preconditions are used so that at each step, only one nondeterministic choice is made. ■

Theorem 5.9 PLAN EXISTENCE *is* EXPSPACE*-complete. It is still* EXPSPACE*-complete if we restrict* **P** *to be positive.*

Synopsis of proof. We define a reduction from EXPSPACE Turing machine acceptance problem. We use predicates contains($\underline{j}, c$), head($\underline{t}$), state(q) to denote that at the current configuration, cell $\underline{j}$ contains c, head is at position $\underline{t}$ and state is q. Operations mimic the "next move" function; e.g., moving right deletes h($\underline{t}$) and adds h($\underline{t+1}$). ■

We show below that when we restrict preconditions of planning operators to contain at most one *atom*, then the planning problem is PSPACE-complete.

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

Synopsis of proof. To prove hardness, we define a reduction from the linearly bounded automata (LBA) acceptance problem. We use a predicate of arity $n + 2$ to represent a machine configuration. Operators with one predicate in the add-list and one positive precondition suffice to mimic the "next move" function. To prove membership we use the fact that each operator has at most one precondition. ■

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

Synopsis of proof. PLAN EXISTENCE is a special case of PLAN LENGTH, with k = the number of ground instances of predicates. PLAN EXISTENCE was proved to be PSPACE-hard (Theorem 5.10), hence hardness follows. For proving membership, we use the same algorithm we used to determine PLAN EXISTENCE in Theorem 5.10, but we modify it to keep track of the number of actions, and fail if this number exceeds k. ■

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

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

Synopsis of proof. We define a polynomial reduction from the NEXPTIME Turing machine acceptance problem to case 1. Since case 1 is a special case of 2, 3, and 4, this proves that all three cases are NEXPTIME-hard. To prove that they are in NEXPTIME, notice that plan length is part of the input. We can nondeterministicaly choose a sequence of k or less operator instances and verify that it is a valid plan. ∎

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. 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 no predicate/proposition that does not appear in the operators appears in G or S_0. Since the operators can neither add nor delete atoms constructed from these predicates, this is a reasonable restriction.

5.3.1 Propositional Planning

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 Function-free First-Order Planning

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* **P** *to be fixed and deletion-free, then* PLAN EXISTENCE *and* PLAN LENGTH *are in* NP.

4. *If we restrict* **P** *to be fixed, then* PLAN EXISTENCE *and* PLAN LENGTH *are in* PSPACE.

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

The above theorem puts a bound on how hard planning can be with a fixed set of operators. The following theorems state that we can find fixed sets of operators such that their corresponding planning problems are complete for these complexity classes.

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

Synopsis of proof. We show how to reduce the satisfiability problem to a fixed set of operators. We make use of the bound on the length of the plan to ensure each boolean variable is assigned a unique truth value. ■

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

Synopsis of proof. PLAN LENGTH follows from theorem5.15. For PLAN EXISTENCE, we present a fixed set of operators that satisfiability problem can be reduced to as the proof. We make use of negated preconditions to ensure each boolean variable is assigned a unique truth value. ■

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

Synopsis of proof. We give a fixed set of operators that mimic an LBA. ■

6 Conclusion

The primary aim of this paper has been to develop an exhaustive analysis of the complexity and decidability of planning with STRIPS-like planning operators (i.e., operators comprised of preconditions, add lists, and delete lists).

One of our primary results is that planning is decidable if and only if the language has finitely many ground terms. This relates in several ways to Chapman's work [4]:

1. It solves an open problem posed in [4], regarding the decidability of planning with a finite initial state. The answer depends on whether the language has finitely many ground terms.

2. It shows that one of the results in [4] is misleading. The undecidability of first-order planning has nothing to do with the presence or absence of conditional effects. If the language has finitely many ground terms, then planning is decidable even if the operators have conditional effects. If the language has infinitely many ground terms, then planning is undecidable even if no operators have no conditional effects, no delete lists, and no negative preconditions.

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 1. Examination of this table reveals several interesting properties.

1. Comparing the complexity of PLAN EXISTENCE in the first-order case with the corresponding propositional case reveals a regular pattern: in most cases, the complexity of the first-order problem is exactly one level harder than the complexity of the corresponding propositional planning problem. We have EXPSPACE-complete versus PSPACE-complete, NEXPTIME-complete versus NP-complete, EXPTIME-complete versus polynomial.

2. For first-order function-free planning, 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 propositional planning, because the length 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 help PLAN LENGTH problem. As a result, for propositional planning, both PLAN EXISTENCE and PLAN LENGTH are PSPACE-complete.

3. When the operator set is fixed in advance, each first-order operator can be mapped into a set of propositional operators. Thus, first-order planning with a fixed set of operators has basically the same complexity as propositional planning with the operators given as part of the input.

4. 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.

In summary, planning is a hard problem even under severe restrictions on the nature of planning domains. This paper explains precisely how the complexity of planning increases as we allow more general planning operators to be used.

References

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

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

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

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

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

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

[7] Naresh Gupta and Dana S. Nau. "Complexity results for blocks-world planning," *Proc. AAAI-91*, 1991. Honorable mention for the best paper award.

[8] Naresh Gupta and Dana S. Nau, "On the complexity of blocks world planning," submitted for publication, 1991.

[9] 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.

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

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

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

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

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

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

[16] 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.

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

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

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

[20] 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*, pages 26–49. Morgan Kaufman, 1990.

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

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

Appendix: Proofs of Theorems and Lemmas

Proof of Lemma 3.1.

1. Immediate consequence of the fact that $\mathrm{Del}(\alpha) = \emptyset$ for all $\alpha \in \mathcal{O}$. Hence, for all $0 \leq i \leq n$,
$$S_{i+1} = S_i \cup \mathrm{Add}(\alpha_i)\theta_i.$$

2. Suppose α is θ-executable in state S_j.Then $\mathrm{Pre}(\alpha)\theta \subseteq S_j \subseteq S_k$ is true. As $\mathrm{Pre}(\alpha)$ is negation free, the condition that $\{B\theta \mid \neg B$ is a negated atom in $\mathrm{Pre}(\alpha)\} \cap S_k = \emptyset$ is immediately satisfied and hence α is executable in state S_k. This completes the proof.

■

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, if I is a state, then $T_P(I)$ is the set of all ground atoms that can be proved in one step by using the rules of program P as well as by taking the atoms in I to be true.

Definition 6.1 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: $T_P(I) = \{A \mid A$ 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.

Proof of Theorem 3.1 (Equivalence Theorem). 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

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

is a planning sequence such that $A_1\sigma \,\&\, \ldots \,\&\, A_s\sigma$ is true in S_{r+1}. 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+1} by Lemma 3.1. Likewise, every $A_{\rho(j)} \in X_2$ is true in $S_{\rho(j)} \subseteq S_{r+1}$ by Lemma 3.1.

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

$$S_0 \overset{\alpha_{v(0)},\theta_{v(0)}}{\Longrightarrow} S_1^1 \overset{\alpha_{v(1)},\theta_{v(1)}}{\Longrightarrow} \cdots \overset{\alpha_{v(k_1)},\theta_{v(k_1)}}{\Longrightarrow} S_{k_1}^1$$

and $\wp_2$ is the sequence

$$S_0 \overset{\alpha_{w(0)},\theta_{w(0)}}{\Longrightarrow} S_1^2 \overset{\alpha_{w(1)},\theta_{w(1)}}{\Longrightarrow} \cdots \overset{\alpha_{w(k_2)},\theta_{w(k_w)}}{\Longrightarrow} S_{k_2}^2.$$

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

$$\begin{array}{lll}
S_0 & \overset{\alpha_{v(0)},\theta_{v(0)}}{\Longrightarrow} & S_1^1 \\
& \overset{\alpha_{w(0)},\theta_{w(0)}}{\Longrightarrow} & S_1^1 \cup S_1^2 \\
& \overset{\alpha_{v(1)},\theta_{v(1)}}{\Longrightarrow} & S_1^1 \cup S_1^2 \cup S_2^1 \\
& \overset{\alpha_{w(1)},\theta_{w(1)}}{\Longrightarrow} & S_1^1 \cup S_1^2 \cup S_2^1 \cup S_2^2 \\
& \cdots & \\
& \overset{\alpha_{v(k_1)},\theta_{v(k_1)}}{\Longrightarrow} & \left(\bigcup_{j=1}^{k_1} S_j^1\right) \cup \left(\bigcup_{z=1}^{k_2-1} S_z^2\right) \\
& \overset{\alpha_{w(k_2)},\theta_{w(k_w)}}{\Longrightarrow} & \left(\bigcup_{j=1}^{k_1} S_j^1\right) \cup \left(\bigcup_{z=1}^{k_2} S_z^2\right).
\end{array}$$

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_i^1 and hence, by Lemma 3.1, it is also $\theta_{v(i)}$-executable in the state $\bigcup_{j=1}^{i} S_j^1 \cup \bigcup_{h=1}^{i-1} S_h^2$. The same reasoning applies to the $\alpha_{w(j)}$'s.
2. As $A_1\sigma \in S_{k_1}^1$ and as $A_2\sigma \in S_{k_2}^2$, it follows that $\{A_1\sigma, A_2\sigma\} \subseteq \left(\bigcup_{j=1}^{k_1} S_j^1\right) \cup \left(\bigcup_{z=1}^{k_2} S_z^2\right)$.

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 $(A_1\sigma \,\&\, \ldots \,\&\, A_s\sigma)$.

($\Longleftarrow$) : 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:

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

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

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

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

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

$$(\forall)(H \leftarrow \&_{K \in \text{Pre}(\alpha_t)} K).$$

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

■

Proof of Corollary 3.1 (Semi-Decidability Results). Immediate consequence of Theorem 3.1 and a result of Blair which shows that all recursively enumerable sets of ground atoms can be represented as the set of ground atoms provable from a logic programs[1]. ■

Proof of Corollary 3.2. Immediate consequence of Theorem 3.1 and a result of Blair [1] which shows that the class of determinate logic programs ([1]) is Π^0_2-complete. ■

Proof of Corollary 3.3. Immediate consequence of Theorem 3.1 and a result of Sebelik and Stepanek [16] 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. ■

Proof of Lemma 3.2.

1. As G is a goal, G is of the form

$$(\exists)(A_1 \,\&\, \ldots \,\&\, A_k).$$

Let $G_1, G_2, \ldots$ be an arbitrary enumeration of all ground instances of $(A_1 \,\&\, \ldots \,\&\, A_k)$. G is true in the Herbrand interpretation I iff $(\exists i)G_i$ is true in I. As G_i is a conjunction of ground atoms, the problem "is G_i true in I ?" is decidable. Hence, the problem "does there exists an i such that G_i true in I ?" is semi-decidable.

2. In this case, as $\mathcal{L}$ is function-free, there are only finitely many ground instances of $(A_1 \,\&\, \ldots \,\&\, A_k)$. Let $G_1, \ldots, G_r$ be these ground instances. Checking if a specific G_i is true in I is decidable. Thus, checking if one of the G_i's, $1 \leq i \leq k$, is true in I is decidable because we simply need to check the G_i's one by one.

■

Proof of Theorem 3.2. As $\mathbf{P}$ 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. ■

Proof of Theorem 3.3 (Theorem on Infinite Initial States). 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. ■

Proof of Theorem 3.4 (Eliminating Delete Lists). 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 predicate symbol "poss" of arity one.

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}, $1 \leq \mathrm{init} \leq 2^k$, denotes the initial state S_0 of $\mathbf{P}$. 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

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

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

$$\begin{aligned} \text{Pre}: &\quad \{\text{poss}(s_i)\} \cup \{p(t_1,\ldots,t_n,s_i) | p(t_1,\ldots,t_n) \in S_i\} \\ \text{Add}: &\quad \{\text{poss}(s_j)| \cup \{p(t_1,\ldots,t_n,s_j) | p(t_1,\ldots,t_n) \in S_j\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

(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 $\mathbf{P}'$. ■

Proof of Theorem 3.5 (Decidability of Function-Free Planning). Immediate consequence of Theorems 3.4 and 3.2. ■

Proof of Theorem 5.1. 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$(\ldots)$ has arity kv.

start() 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:

$$\begin{aligned} \text{Pre}: &\quad S_1 \cup S_2 \cup \{\text{start}()\} \\ \text{Del}: &\quad \text{Del}(O) \cup \{p' | p \in \text{Add}(O)\} \\ \text{Add}: &\quad \text{Add}(O) \cup \{p' | p \in \text{Del}(O)\} \end{aligned}$$

where S_1 is the set of all nonnegated atoms in Pre(O), and S_2 is the set of complementary predicates corresponding to the negated atoms in 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 5.1.1.

$$\begin{aligned} \text{Pre}: &\quad \{\text{counter}(\underline{i})\} \\ \text{Del}: &\quad \{\text{counter}(\underline{i})\} \\ \text{Add}: &\quad \{\text{counter}(\underline{i+1})\} \cup \{p'(\underline{i+1}) | p \in P\} \end{aligned}$$

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

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.

■

Proof of Theorem 5.3. 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 preconditon. 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 a 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. ■

Proof of Theorem 5.4. 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.

We define a polynomial reduction from the Set Cover problem (SC) to prove that our planning problem is NP-hard.

SC is defined as follows : Given a set S, a set C which is a collection of subsets of S, and a positive integer k encoded in binary, does there exist a subset C' of C of size at most k, such that each element of S appears in some set in C'. SC is known to be NP-complete (Gaarey and Johnson [5]) .

Here is the transformation:

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. ■

Proof of Theorem 5.5.

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. ■

Proof of Theorem 5.7.

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.Besides, 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. Since there are only an exponential number of operator instances in terms of the input length, the algorithm halts in exponential time. Hence 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 tuple (s, q, j). A configuration is an accepting configuration if the state is an existential state, and there exists a move in δ that leads the configuration to an accepting one, or if the state is universal and all moves lead to an accepting configuration. By definition, a universal configuration with no possible moves is an accepting one.

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: 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 :	$U_i^{q,A}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$

$$\begin{aligned} \text{Pre}: &\quad \{\text{Pre}_L \cup \text{Pre}_R \cup \text{Pre}_S\} \\ \text{Del}: &\quad \emptyset \\ \text{Add}: &\quad \{\text{accept}(v_1, \ldots, v_{i-1}, A, v_{i+1}, \ldots, v_n, q, p_i)\} \end{aligned}$$

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 $i = n$.

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. ∎

Proof of Theorem 5.8. 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 an 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 5.1.1.

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

Name :	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. ■

Proof of Theorem 5.9. 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.

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

> INSTANCE: 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.
> OUTPUT: Yes, if M accepts the string x, no otherwise.

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 5.1.1.

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. ■

Proof of Theorem 5.10. 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 [5]) 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 $p_i, i \neq 1$:

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

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

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

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

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

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. ■

Proof of Theorem 5.11. 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 $Q \cap \text{Add}(\alpha) \neq \emptyset$.
4. Decrement counter. Halt and reject if counter=0.
5. $Q := (Q - \text{Add}(\alpha)) \cup (\text{Pre}(\alpha) - S_0)$.
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.

■

Proof of Theorem 5.12.

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, of length at most k, and check that it is a plan in exponential time. 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 instance:

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(...) 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(...) 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 5.1.1.

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. ■

Proof of Theorem 5.13. 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. ■

Proof of Corollary 5.5. Since the number of possible plans is constant, we can check all of them in constant time. ■

Proof of Theorem 5.14. 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. ■

Proof of Theorem 5.15. Here are the operators:

$$\begin{aligned} \text{Pre}: &\quad \{\text{counter}(X), \text{next}(X, Y)\} \\ \text{Add}: &\quad \{\text{counter}(Y), \text{true}(X)\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

$$\begin{aligned} \text{Pre}: &\quad \{\text{counter}(X), \text{next}(X, Y)\} \\ \text{Add}: &\quad \{\text{counter}(Y), \text{false}(X)\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

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

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

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. ■

Proof of Theorem 5.16. NP-hardness for PLAN LENGTH follows from Theorem 5.15. Here are the operators for which PLAN EXISTENCE is NP-hard:

$$\begin{aligned} \text{Pre}: &\quad \{\neg\text{true}(X)\} \\ \text{Add}: &\quad \{\text{false}(X)\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

$$\begin{aligned} \text{Pre}: &\quad \{\neg\text{false}(X)\} \\ \text{Add}: &\quad \{\text{true}(X)\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

$$\begin{aligned} \text{Pre}: &\quad \{\text{poslit}(X,C), \text{true}(X)\} \\ \text{Add}: &\quad \{\text{done}(C)\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

$$\begin{aligned} \text{Pre}: &\quad \{\text{neglit}(X,C), \text{false}(X)\} \\ \text{Add}: &\quad \{\text{done}(C)\} \\ \text{Del}: &\quad \emptyset \end{aligned}$$

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 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. ■

Proof of Theorem 5.17. Here is the set of operators:

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

$$\text{Name}: \quad L(I,J,V,Q,S,Y)$$

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 linearly bound 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

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

The operators mimic the moves of the LBA one to 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. ■

Proof of Theorem 5.6. 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 $\mathbf{P}$ iff there exists a plan to achieve G in $\mathbf{P}'$. ■