Computational Complexity of Planning Based on Partial Information About The System's Present and Past States

Chitta Baral[1], Le Chi Tuan[1], Raul Trejo[2], and Vladik Kreinovich[2]

[1] Dept. of Computer Science & Engineering
Arizona State University, Tempe, AZ 85287-5406, USA,
{chitta,lctuan}@asu.edu
[2] Department of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA, {rtrejo,vladik}@cs.utep.edu

Abstract. Planning is a very important AI problem, and it is also a very time-consuming AI problem. To get an idea of how complex different planning problems are, it is useful to describe the computational complexity of different general planning problems. This complexity has been described for problems in which planning is based on the (complete or partial) information about the *current* state of the system. In real-life planning, in addition to this information, we often also use the knowledge about the system's *past* behavior. To describe such more realistic planning situations, a special language $\mathcal{L}$ was developed in 1997 by C. Baral, M. Gelfond and A. Provetti. In this paper, we expand the known results about computational complexity of planning (including our own previous results about complexity of planning in the planning language $\mathcal{A}$) to this more general class of planning problems.

1 Introduction

It is important to analyze computational complexity of planning problems. Planning is one of the most important AI problems, but it is also known to be one of the most difficult ones. While often in practical applications, we need the planning problems to be solved within a reasonable time, the actual application of planning algorithms may take an extremely long time. It is therefore desirable to estimate the potential computation time which is necessary to solve different planning problems, i.e., to estimate the *computational complexity* of different classes of planning problems. Even "negative" results, which show that the problem belongs to one of the high-level complexity classes (e.g., that it is **PSPACE**-hard) are potentially useful: first, they prevent researchers from wasting their time on trying to design a general efficient algorithm; second, they enable the researchers to concentrate on either finding a feasible sub-class of the original class of planning problems, or on finding (and/or justifying) an approximate planning algorithm.

Known computational complexity results: in brief. There have been several results on computational complexity of planning problems. These results mainly cover the situations in which we have a (complete or partial) information about the current state of the system, and we must find an appropriate plan (sequence of actions) which would enable us to achieve a certain goal. Such situations are described by the language $\mathcal{A}$ which was proposed in [7] (the complexity of planning in $\mathcal{A}$ was analyzed in our earlier paper [3]). In this language, we start with a finite set of properties (fluents) $\mathcal{F} = \{f_1, \ldots, f_n\}$ which describe possible properties of a state. A *state* is then defined as a finite set of fluents; e.g., if $n = 2$, then the state $\{f_1\}$ means a state in which f_1 is true, and f_2 is not. Our knowledge about the initial state is described by statements of the type "initially F", where F is a *fluent literal*, i.e., a fluent or its negation. There is also a finite set $\mathcal{A}$ of possible *actions*. The results of different actions $a \in \mathcal{A}$ are described by rules of the type "a causes F if $F_1, \ldots, F_m$", where $F, F_1, \ldots, F_m$ are fluent literals. The semantics include "inertia": if an action does not lead to f or $\neg f$, then the value of the fluent f does not change.

To formulate a planning problem, we must select an *objective*. In general, our objective can be an arbitrary logical combination of the basic properties (fluents); however, since we can always add this combination as a new fluent, we can, without losing generality, assume that our objective is one of the fluents g from $\mathcal{F}$. A *plan* is a sequence of actions $\alpha = [a_1, \ldots, a_m]$. We say that a plan is *successful* if for every initial state s which is consistent with our knowledge, after we apply the plan α, the desired fluent g holds in the resulting state $res(\alpha, s)$.

Ideally, we want to find cases in which the planning problem can be solved by a *feasible* algorithm, i.e., by an algorithm $\mathcal{U}$ whose computational time $t_{\mathcal{U}}(w)$ on each input w is bounded by a polynomial $p(|w|)$ of the length $|w|$ of the input w: $t_{\mathcal{U}}(x) \leq p(|w|)$ (this length can be measured bit-wise or symbol-wise). Since, in practice, we are operating in a time-bounded environment, we should worry not only about the time for *computing* the plan, but we should also worry about the time that it takes to actually *implement* the plan. If an action plan consists of a sequence of 2^{2^n} actions, then this plan is not feasible. It is therefore reasonable to restrict ourselves to *feasible* plans, i.e., by plans u whose length m (= number of actions in it) is bounded by a given polynomial $p(|w|)$ of the length $|w|$ of the input w. For each such polynomial p, we can formulate the following *planning problem*: given a domain description D (i.e., the description of the initial state and of possible consequences of different actions) and a goal g (i.e., a fluent which we want to be true), determine whether it is possible to feasibly achieve this goal, i.e., whether there exists a feasible plan α (with $m \leq p(|D|)$) which achieves this goal.

By solving this problem, we do not yet get the desired plan, we only check whether a plan exists. However, intuitively, the complexity of this problem also represents the complexity of actually finding a plan, in the following sense: if we have an algorithm which solves the above planning problem in reasonable time, then we can also find this plan. Indeed, suppose that we are looking for a plan of length $m \leq P_0$, and an algorithm has told us that such a plan exists.

Then, to find the first action of the desired plan, we check (by applying the same algorithm), for each action $a \in \mathcal{A}$, whether from the corresponding state $res(a, s)$ the desired goal g can be achieved in $\leq P_0 - 1$ steps. Since a plan of length $\leq P_0$ does exist, there is such an action, and we can take this action as a_1. After this, we repeat the same procedure to find a_2, etc. As a result, we will be able to find a plan of length $\leq P_0$ by applying the algorithm which checks the existence of the plan $\leq P_0 = p(|D|)$ times; so, if the existence-checking algorithm is feasible, the resulting plan-construction algorithm is feasible as well.

General results on computational complexity of planning are given, e.g., in [4, 6, 10]. For the language $\mathcal{A}$, computational complexity of planning was first studied in [9]; the results about the computational complexity of different planning problems in $\mathcal{A}$ are overviewed in [3, 14].

If the initial information is incomplete, then, in addition to normal actions which form the plan, it is reasonable to consider *sensing* actions, i.e., actions which may not change the state of the system, but which enable us to find the missing information. To describe such actions, the language $\mathcal{A}$ was enriched by rules of the type "a determines f", meaning that after the action a is performed, we know whether f is true or not. At any given moment of time, we have the actual state s of the system (which may be not completely known to the agent), plus a set Σ of all possible states which are consistent with the agent's knowledge; the pair $\langle s, \Sigma \rangle$ is called a *k-state*. A sensing action does not change the actual state s, but it does decrease the set Σ.

In planning, the main purpose of sensing actions is to make a planning decision depending on the actual value of the sensed fluent. Thus, when sensing is allowed, a plan is not a sequence, but rather a *tree*: every sensing action means that we branch into two possible branches (depending on whether the sensed fluent is true or false), and we execute different actions on different branches. Similarly to the case of the linear plan, we are only interested in plans whose execution time is (guaranteed to be) bounded by a given polynomial $p(|D|)$ of the length of the input. (In other words, we require that for every possible branch, the total number of actions on this branch is bounded by $p(|D|)$.)

For such planning situations, the computational complexity was also surveyed in [3].

Towards a more realistic formulation of planning problems. The planning problem, as formulated in the language $\mathcal{A}$, is based on the assumption that the only information we have is the information about the current state. In real life, in addition to the information about the current state, we often have some information about the previous behavior of the system.

To describe such more realistic planning problems, in [1, 2], the language $\mathcal{A}$ was extended to a new language $\mathcal{L}$. In this new language, to describe the history of the system, first of all, the current state s_N is separated from the initial state s_0, so we may have statements about what is true at s_0 ("F at s_0") and statements about what is true at s_N ("F at s_N"). In addition, we may have information about other states in the past; to describe this information,

language $\mathcal{L}$ allows to use several constants s_i to describe past moments of time, and allow:

- statements of the type "s_1 precedes s_2" which order past moments of time;
- statements of the type "F at s_i" which describe the properties of the system at the past moments of time, and
- statements which describe past actions:
  - "α between s_1, s_2" means that a sequence of actions α was performed at some point between the moments s_1 and s_2, and
  - "α occurs_at s" means that the sequence of actions α was implemented at s.

The semantics of this history description is as follows:

- a *history* is defined as a triple consisting of an initial state s_0, a sequence of actions $\alpha = [a_1, \ldots, a_m]$, and a mapping t which maps each constant s_i from the history description into an integer $t(s_i) \le m$ (meaning the moment of time when this constant actually happened, so $t(s_0) = 0$ and $t(s_N) = m$); for this history, we have, at moments of time $0, 1, \ldots, m$, states $s(0) = s_0$, $s(1) = res(a_1, s(0))$, $s(2) = res(a_2, s(1))$, etc., and s_i is identified with $s(t(s_i))$;
- we say that the history is *consistent* with the given knowledge if all the statements from this knowledge become true under this interpretation;
- we say that the history is *possible* if it is consistent and *minimal* in the sense that no history with a proper subsequence of α is consistent.

In this more realistic situation, we can also ask about the existence of a plan, i.e., a sequence (or tree) of actions with a feasible execution time which guarantees that for all possible current states, after this plan, the objective $g \in \mathcal{F}$ will be satisfied.

In this paper, we answer the following natural question: *How does the addition of history change the computational complexity of different planning problems?*

Comment. In addition to the possibility of describing history, the language $\mathcal{A}$ can also be extended by adding *static causal laws*, which can make the results of an action non-deterministic. This non-determinism may further increase the complexity of the corresponding planning problem; we are planning to analyze this increase in our future work.

Useful complexity notions. Most papers on computational complexity of planning problems classifies these problems to different levels of polynomial hierarchy. For precise definitions of the polynomial hierarchy, see, e.g., [11]. Crudely speaking, a decision problem is a problem of deciding whether a given input w satisfies a certain property P (i.e., in set-theoretic terms, whether it belongs to the corresponding set $S = \{w \mid P(w)\}$).

- A decision problem belongs to the class **P** if there is a feasible (polynomial-time) algorithm for solving this problem.

- A problem belongs to the class **NP** if the checked formula $w \in S$ (equivalently, $P(w)$) can be represented as $\exists u P(u, w)$, where $P(u, w)$ is a feasible property, and the quantifier runs over words of feasible length (i.e., of length limited by some given polynomial of the length of the input). The class **NP** is also denoted by $\Sigma_1\mathbf{P}$ to indicate that formulas from this class can be defined by adding 1 existential quantifier (hence Σ and 1) to a polynomial predicate (**P**).
- A problem belongs to the class **coNP** if the checked formula $w \in S$ (equivalently, $P(w)$) can be represented as $\forall u P(u, w)$, where $P(u, w)$ is a feasible property, and the quantifier runs over words of feasible length (i.e., of length limited by some given polynomial of the length of the input). The class **coNP** is also denoted by $\Pi_1\mathbf{P}$ to indicate that formulas from this class can be defined by adding 1 universal quantifier (hence Π and 1) to a polynomial predicate (hence **P**).
- For every positive integer k, a problem belongs to the class $\Sigma_k\mathbf{P}$ if the checked formula $w \in S$ (equivalently, $P(w)$) can be represented as $\exists u_1 \forall u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where $P(u_1, \ldots, u_k, w)$ is a feasible property, and all k quantifiers run over words of feasible length (i.e., of length limited by some given polynomial of the length of the input).
- Similarly, for every positive integer k, a problem belongs to the class $\Pi_k\mathbf{P}$ if the checked formula $w \in S$ (equivalently, $P(w)$) can be represented as $\forall u_1 \exists u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where $P(u_1, \ldots, u_k, w)$ is a feasible property, and all k quantifiers run over words of feasible length (i.e., of length limited by some given polynomial of the length of the input).
- All these classes $\Sigma_k\mathbf{P}$ and $\Pi_k\mathbf{P}$ are subclasses of a larger class **PSPACE** formed by problems which can be solved by a polynomial-*space* algorithm. It is known (see, e.g., [11]) that this class can be equivalently reformulated as a class of problems for which the checked formula $w \in S$ (equivalently, $P(w)$) can be represented as $\forall u_1 \exists u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where the number of quantifiers k is bounded by a polynomial of the length of the input, $P(u_1, \ldots, u_k, w)$ is a feasible property, and all k quantifiers run over words of feasible length (i.e., of length limited by some given polynomial of the length of the input).

A problem is called *complete* in a certain class if, crudely speaking, this is the toughest problem in this class (so that any other general problem from this class can be reduced to it by a feasible-time reduction).

It is still not known (2000) whether we can solve any problem from the class **NP** in polynomial time (i.e., in precise terms, whether **NP**=**P**). However, it is widely believed that we cannot, i.e., that **NP**$\neq$**P**. It is also believed that to solve a **NP**-complete or a **coNP**-complete problem, we need exponential time $\approx 2^n$, and that solving a complete problem from one of the second-level classes $\Sigma_2\mathbf{P}$ or $\Pi_2\mathbf{P}$ requires more computation time than solving **NP**-complete problems (and solving complete problems from the class **PSPACE** takes even longer).

2 Results

In accordance with the above text and with [3], we will consider the following four main groups of planning situations:

- complete information about the initial state, no sensing actions allowed;
- possibly incomplete information about the initial state, no sensing actions allowed;
- possibly incomplete information about the initial state, sensing actions allowed;
- possibly incomplete information about the initial state, full sensing (i.e., every fluent can be sensed).

For comparison, we will also mention the results corresponding to the language $\mathcal{A}$, when neither history no static causal laws are allowed.

Before we describe the computational complexity of checking the existence of a plan, let us consider a simpler problem: if, through some heuristic method, we have a plan, how can we check that this plan works?

This plan checking problem makes perfect sense only for the case of no sensing: indeed, if sensing actions are possible, then we can have a branching at every step; as a result, the size of the tree can grow exponentially with the plan's execution time, and even if we can check this tree plan in time polynomial in its size, it will still take un-realistically long.

For the language $\mathcal{A}$, the complexity of this problem depends on whether we have complete information of the initial state or not:

Theorem 1. (language $\mathcal{A}$, no sensing)

- *For situations with complete information, the plan checking problem is feasible.*
- *For situations with incomplete information, the plan checking problem is* **coNP***-complete.*

Comment. For readers' convenience, all the proofs are placed in the special (last) section.

Theorem 2. (language $\mathcal{L}$, no sensing)

- *For situations with complete information about the initial state, the plan checking problem is* Π_2**P***-complete.*
- *For situations with incomplete information about the initial state, the plan checking problem is* Π_2**P***-complete.*

Comment. The problem remains Π_2**P**-complete even if we only consider situations with two possible actions. If we only have one action, then for complete information, plan checking is feasible; for incomplete information, it is **coNP**-hard.

Now, we are ready to describe complexity of planning. In the framework of the language $\mathcal{A}$ (i.e., without history), most planning problems turn out to be

complete in one of the classes of the polynomial hierarchy; see, e.g., [3]. However, it turns out that when we allow history, i.e., when we move from language $\mathcal{A}$ to the language $\mathcal{L}$, we get a planning problem that does not seem to be complete within any of the classes from the polynomial hierarchy. To describe the complexity of this program, we therefore had to search for appropriate intermediate classes.

In this search, we were guided by the example of intermediate classes which have been already analyzed in complexity theory: namely, the classes belonging to the so-called *Boolean hierarchy* (see, e.g., [5, 11]). This hierarchy started with the discovery of the first such class – the class **DP** [12, 13]. The original description of these classes uses a language which is slightly different from the language that we used to describe the polynomial hierarchy: namely, we described these classes in terms of the corresponding logical formulas, while the standard description of Boolean hierarchy uses oracles or sets. Therefore, before we explain the new intermediate complexity class which turned out just right for planning, let us first reformulate the notion of the Boolean hierarchy in terms of the corresponding logical formulas.

After $\mathbf{NP}=\Sigma_1\mathbf{P}$ and $\mathbf{coNP}=\Pi_1\mathbf{P}$, the next classes in the polynomial hierarchy are $\Sigma_2\mathbf{P}$ and $\Pi_2\mathbf{P}$. In particular, $\Sigma_2\mathbf{P}$ is a class of problems for which the checked formula $P(w)$ can be represented as $\exists u_1 \forall u_2 P(u_1, u_2, w)$ for some feasible property $P(u_1, u_2, w)$. For each given w, to check whether w satisfies the desired property, we must therefore check whether the following formula holds: $\exists u_1 \forall u_2 Q(u_1, u_2)$, where by $Q(u_1, u_2)$, we denoted $P(u_1, u_2, w)$. In the general definition of this class, for each w, $Q(u_1, u_2)$ can be an arbitrary (feasible) binary predicate. Therefore, in order to find a subclass of this general class $\Sigma_2\mathbf{P}$ for which decision problem is easier than in the general case, we must look for predicates which are simpler than the general binary predicates.

Which predicates are simpler than binary? A natural answer is: unary predicates. It is therefore natural to consider the formulas in which $Q(u_1, u_2)$ is actually a unary predicate, i.e., formulas in which $Q(u_1, u_2)$ depends only on one of its variables. In other words, we have either $Q(u_1, u_2) \equiv Q_2(u_1)$ (here, the subscript 2 in Q_2 means that the predicate does not depend on u_2), or $Q(u_1, u_2) \equiv Q_1(u_2)$. Both these classes of "simpler" binary predicates do lead to simpler complexity classes, but these classes are still within the polynomial hierarchy. Indeed:

- the formula $\exists u_1 \forall u_2 Q_2(u_1)$ is equivalent to $\exists u_1 Q_2(u_1)$ and therefore, the corresponding complexity class is exactly $\Sigma_1\mathbf{P}$ $(= \mathbf{NP})$;
- the formula $\exists u_1 \forall u_2 Q_1(u_2)$ is equivalent to $\forall u_2 Q_1(u_2)$ and therefore, the corresponding complexity class is exactly $\Pi_1\mathbf{P}$ $(= \mathbf{coNP})$.

We get non-trivial intermediate classes if we slightly modify the above idea: namely, if instead of restricting ourselves to binary predicates $Q(u_1, u_2)$ which are actually unary, we consider binary predicates which are Boolean combinations of unary predicates.

For example, we can consider the case when $Q(u_1, u_2)$ is a conjunction of two unary predicates, i.e., when $Q(u_1, u_2)$ is equivalent to

$Q_1(u_2)\&Q_2(u_1)$. In this case, the formula $\exists u_1 \forall u_2 (Q_1(u_2)\&Q_2(u_1))$ is equivalent to $\exists u_1 Q_2(u_1)\&\forall u_2 Q_1(u_2)$. If we explicitly mention the variable w, we conclude that $w \in S$ is equivalent to $\exists u_1 P_2(u_1, w)\&\forall u_2 P_1(u_2, w)$, i.e., that the set S is equal to the intersection of a set $S_1 = \{w \,|\, \exists u_1 P_2(u_1, w)\}$ from the class **NP** and a set $S_2 = \{w \,|\, \forall u_2 P_1(u_2, w)\}$ from the class **coNP**, i.e., equivalently, to the difference $S_1 - (-S_2)$ between two sets S_1 and $-S_2$ (a complement to S_2) from the class **NP**. Such sets represent the difference class **DP**, the first complexity class from the Boolean hierarchy. If we allow more complex Boolean combinations of unary predicates, we get other complexity classes from this hierarchy.

For planning, we need a simpler subclass within the class $\Sigma_3\mathbf{P}$ of all formulas $P(w)$ of the type $\exists u_1 \forall u_2 \exists u_3 P(u_1, u_2, u_3, w)$. Similarly to the above description of the Boolean hierarchy, it is natural to consider the cases when, for every w, the corresponding ternary predicate $P(u_1, u_2, u_3, w)$ (for fixed w) can be represented as a Boolean combination of binary predicates $P_1(u_2, u_3, w)$, $P_2(u_1, u_3, w)$, and $P_3(u_1, u_2, w)$. Let us give a formal definition of such classes.

Definition. *Let $k \geq 1$ be an integer. By a k-marked propositional variable, we mean an expression of the type v^j, where v is a variable and j is an integer from 1 to k. By a k-Boolean expression B, we mean a propositional formula $B(v_1^{j_1}, \ldots, v_m^{j_m})$ in which all variables are k-marked.*

- *For every k-Boolean expression B, by a class $\Sigma_k(B)\mathbf{P}$, we mean the class of all problems for which the checked formula $P(w)$ can be represented as $\exists u_1 \forall u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where $P(u_1, \ldots, u_k, w) = B(P_1, \ldots, P_m)$, and for each i, the corresponding predicate P_i is feasible and does not depend on the variable u_{j_i}.*
- *For every k-Boolean expression B, by a class $\Pi_k(B)\mathbf{P}$, we mean the class of all problems for which the checked formula $P(w)$ can be represented as $\forall u_1 \exists u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where $P(u_1, \ldots, u_k, w) = B(P_1, \ldots, P_m)$ and for each i, the corresponding predicate P_i is feasible and does not depend on the variable u_{j_i}.*

For example, the above class **DP** can be represented as $\Sigma_2(v^1\&v^2)\mathbf{P}$.

Theorem 3. (language $\mathcal{L}$, no sensing) *For situations with complete information about the initial state and with no sensing, the computational complexity of planning is $\Sigma_3(v^1 \vee v^3)\mathbf{P}$-complete.*

Comments.

- In other words, the corresponding planning problem is complete for the class of all problems in which $P(w)$ is equivalent to $\exists u_1 \forall u_2 \exists u_3 (P_1(u_2, u_3) \vee P_3(u_1, u_2))$.
- The fact that the planning problem is complete for an intermediate complexity class is not surprising: e.g., in [8], it is shown that several planning problems are indeed complete in some classes intermediate between standard classes of polynomial hierarchy.

- The problem remains $\Sigma_3(v^1 \vee v^3)\mathbf{P}$-complete even if we only consider situations with two possible actions. If we only have one action, then for complete information, planning is feasible; for incomplete information, it is **coNP**-hard.
- For $\mathcal{A}$, the corresponding planning problem is **NP**-complete.

Theorem 4. (language $\mathcal{L}$, no sensing) *For situations with incomplete information about the initial state and with no sensing, the computational complexity of planning is $\Sigma_3(v^1 \vee v^3)\mathbf{P}$-complete.*

For $\mathcal{A}$, this problem is $\Sigma_2\mathbf{P}$-complete.

Theorem 5. (language $\mathcal{L}$, with sensing) *For situations with incomplete information about the initial state and with sensing, the computational complexity of planning is* **PSPACE***-complete.*

For $\mathcal{A}$, this problem is also **PSPACE**-complete.

Theorem 6. (language $\mathcal{L}$, full sensing) *For situations with incomplete information about the initial state and with full sensing, the computational complexity of planning is $\Pi_2\mathbf{P}$-complete.*

For $\mathcal{A}$, this problem is also $\Pi_2\mathbf{P}$-complete.

3 Proofs

Proof of Theorem 1. Theorem 1 is, in effect, proven in [3].

Proof of Theorem 2. Let us first show that the plan checking problem belongs to the class $\Pi_2\mathbf{P}$. Indeed, a given plan w is successful if it succeeds for every possible history u_1. For every given history u_1, checking whether a given plan w succeeds is feasible; we will denote the corresponding predicate by $S(u_1, w)$. The condition that the history u_1 is possible means that it is consistent and that none of its sub-histories u_2 is consistent. Checking consistency is feasible (we will denote the corresponding predicate by $C(u)$), and checking whether u_2 is a consistent sub-history of the history u_1 is also feasible; we will denote this other predicate by $H(u_1, u_2)$. So, the possibility of a history u_1 can be expressed as $C(u_1)\&\neg\exists u_2 H(u_1, u_2)$, which is equivalent to $\forall u_2(C(u_1)\&\neg H(u_1, u_2))$. Hence, the success of the plan w can be expressed as

$$\forall u_1(\forall u_2(C(u_1)\&\neg H(u_1, u_2)) \to S(u_1, w)),$$

i.e., as a formula $\forall u_1 \exists u_2(\neg C(u_1) \vee H(u_1, u_2) \vee S(u_1, w))$ from the class $\Pi_2\mathbf{P}$. So, the plan checking problem indeed belongs to the class $\Pi_2\mathbf{P}$.

Let us prove that the plan checking problem is $\Pi_2\mathbf{P}$-complete. To show it, we will prove that the known $\Pi_2\mathbf{P}$-complete problem – namely, the problem of checking, for a given propositional formula F, whether $\exists x_1 \ldots \exists x_m \forall x_{m+1} \ldots \forall x_n\, F(x_1, \ldots, x_n)$ – can be reduced to plan checking. We will do this reduction for the case when we have a complete information about

the initial state; then, it will automatically follow that a more general problem – corresponding to a case when we may only have partial information about the initial state – is also $\Pi_2\mathbf{P}$-complete.

Let us show how, for each propositional formula F, we can design a plan checking problem whose correctness is equivalent to validity of the above quantified formula. In this problem, we have three actions: a, a^-, and a_0. We have fluents $f_1, \ldots, f_n, \ldots, f_N$ which will correspond to the variables $x_1, \ldots, x_n$ and to auxiliary propositional variables $x_{n+1}, \ldots, x_N$ which we will introduce later. We have fluents $t_0, t_1, \ldots, t_n, \ldots, t_{N+2}$ so that a fluent t_i is true if we are at i-th step of the process; between t_m and t_{n+1}, we also have fluents $t_{m+1/2}, t_{m+3/2}, \ldots, t_{n+1/2}$, which describe "intermediate" steps. We also have two special fluents: a "goal" fluent g, and an auxiliary fluent x. *Initially*, t_0 and g are true, and all other fluents are false. The only information we have about the *current* state is that in this state, x is true and t_{N+2} is true. No other information about history is known.

The plan whose correctness we are checking is an empty sequence of actions; our objective is the goal fluent g. In other words, we are checking whether for every possible history, g holds in the resulting current state.

Let us now enumerate the rules which describe the results of actions. We start with all variables $f_1, \ldots, f_n, \ldots$ being false. The first group of rules describes the possibility of changing, in the first m moments of time, the values of the first m fluents $f_1, \ldots, f_m$. On i-th step ($i = 1, \ldots, m$), the action a makes f_i true, while action a^- keeps f_i false. Whatever action we choose, we go to the next step, i.e., we make t_{i-1} false and t_i true. These rules are as follows:

$$a \text{ causes } f_i \text{ if } t_{i-1}; \quad a \text{ causes } t_i \text{ if } t_{i-1}; \quad a \text{ causes } \neg t_{i-1} \text{ if } t_{i-1};$$

$$a^- \text{ causes } \neg f_i \text{ if } t_{i-1}; \quad a^- \text{ causes } t_i \text{ if } t_{i-1}; \quad a^- \text{ causes } \neg t_{i-1} \text{ if } t_{i-1}.$$

At moment t_m, we progress by applying the special action a_0; the corresponding rules are:

$$a_0 \text{ causes } t_{m+1} \text{ if } t_m; \quad a_0 \text{ causes } \neg t_m \text{ if } t_m.$$

At moment t_{m+1}, we can apply the action a_0 one more time and make g false and x true; the corresponding rules are

$$a_0 \text{ causes } \neg g \text{ if } t_{m+1}; \quad a_0 \text{ causes } x \text{ if } t_{m+1}.$$

Alternatively, we can also skip this second application of the action a_0; in both cases, we remain at the same $(m+1)$-th stage t_{m+1}.

To describe the following evolution, we describe two sets of rules: rules covering the case when x is true, and rules covering the case when x is false.

When x is true, we have a straightforward evolution until moment t_{n+1}; here, $i = m+1, \ldots, n$:

$$a \text{ causes } t_{i+1/2} \text{ if } t_i, x; \quad a \text{ causes } \neg t_i \text{ if } t_i, x;$$

$$a^- \text{ causes } t_{i+1} \text{ if } t_{i+1/2}, x; \quad a^- \text{ causes } \neg t_{i+1/2} \text{ if } t_{i+1/2}, x.$$

When x is false, then we cannot access intermediate steps, and at each whole step, we use a or a^- to pick a value of the corresponding variable f_i, $m+1 \le i \le n$:

$$a \text{ causes } f_i \text{ if } t_i, \neg x; \quad a \text{ causes } t_{i+1} \text{ if } t_i, \neg x; \quad a \text{ causes } \neg t_i \text{ if } t_i, \neg x;$$

$$a^- \text{ causes } \neg f_i \text{ if } t_i, \neg x; \quad a^- \text{ causes } t_{i+1} \text{ if } t_i, \neg x; \quad a^- \text{ causes } \neg t_i \text{ if } t_i, \neg x.$$

The final set of rules corresponds to checking the validity of the propositional formula F for the values $f_1, \ldots, f_n$. The formula $F(x_1, \ldots, x_n)$ is obtained from the Boolean variables $x_1, \ldots, x_n$ by using propositional operations & ("and"), $\vee$ ("or"), and $\neg$ ("not"). In order to describe the remaining rules, let us describe, step by step, how the computer will do this checking. In other words, let us *parse* the formula F. Let us denote the intermediate results of this computation by $x_{n+1}, x_{n+2}, \ldots$ For example, if F is the formula $(x_1 \vee x_2)\&(x_1 \vee \neg x_2)$, then a possible parsing of this formula is as follows:

- we start with the values x_1 and x_2;
- then, we compute the first disjunction $x_3 := x_1 \vee x_2$;
- then, we compute the negation $x_4 := \neg x_2$;
- after that, we are ready to compute the second disjunction $x_5 := x_1 \vee x_4$;
- finally, we compute the truth value of the resulting formula as the conjunction of the two disjunctions: $x_6 := x_3 \& x_5$.

In general, we start with the variables $x_1, \ldots, x_n$, and then, for $k = n+1, n+2, \ldots, N$, we compute the value of x_k in one of the three possible ways:

- either as $x_k := x_{a(k)} \& x_{b(k)}$ for some values $a(k) < k$ and $b(k) < k$;
- or as $x_k := x_{a(k)} \vee x_{b(k)}$ for some values $a(k) < k$ and $b(k) < k$;
- or as $x_k := \neg x_{a(k)}$ for some value $a(k) < k$.

The value x_N is the truth value of the formula F. For every k from $n+1$ to N, we add the rules "a causes t_{k+1} if t_k", "a causes $\neg t_k$ if t_k", and also, depending on which operation computes x_k, the following rules:

- if $x_k := x_{a(k)} \& x_{b(k)}$, we add "$a$ causes f_k if $t_k, f_{a(k)}, f_{b(k)}$";
- if $x_k := x_{a(k)} \vee x_{b(k)}$, we add "$a$ causes f_k if $t_k, f_{a(k)}$" and "a causes f_k if $t_k, f_{b(k)}$";
- if $x_k := \neg x_{a(k)}$, we add "a causes f_k if $t_k, \neg f_{a(k)}$".

Due to these rules, the new value of the fluent f_k is exactly the right propositional combination of the values of the corresponding fluents $f_{a(k)}$ and $f_{b(k)}$.

Finally, to cover the last step, we have rules which make the auxiliary variable x true if x_N was true at moment t_{N+1}:

$$a \text{ causes } x \text{ if } t_{N+1}, \neg f_N; \quad a \text{ causes } t_{N+2} \text{ if } t_{N+1}; \quad a \text{ causes } \neg t_{N+1} \text{ if } t_{N+1}.$$

By definition, a history starting with a given initial state is consistent if it leads to t_{N+2} and x both being true. In the initial state, x was false; there are only two rules which make x true: the rule which uses the second action a_0 on the step t_{m+1}, and the rule which uses f_N on the last step. So, one of these rules had to be used.

Due to our choice of rules, the only way to go from the initial state in which t_0 is true and all other variables t_i are false to the current state in which t_{N+2} is true is to go step-by-step:

- At first, we have to go through the states in which, correspondingly, $t_1, \ldots, t_m$ are true; for that, we have to apply a sequence of m actions, each of which is either a or a^-. We will denote this sequence of actions by α_1.
- Then, we have to go to the state in which t_{m+1} is true; for that, we have to apply the action a_0.
- Then, we may apply a_0 for the second time or we may not. After that:
  - if we apply the action a_0 twice, we have to go through the states in which, correspondingly, $t_{m+1/2}, t_{m+1}, \ldots, t_{n-1/2}, t_n$ are true; for that, we have to apply a sequence of $2(n-m)$ actions $a, a^-, a, a^-, \ldots, a, a^-$; we will denote this sequence of actions by α_2^+;
  - if we do not apply the action a_0 at the moment t_{m+1}, then we have to go through the states in which, correspondingly, $t_{m+1}, \ldots, t_{n-1}, t_n$ are true; for that, we have to apply a sequence of $n-m$ actions each of which is either a or a^-; we will denote this sequence of actions by α_2^-.
- Finally, we have to go through the states in which, correspondingly, $t_{n+1}, \ldots, t_{N+2}$ are true; the only way to do that is to use $N+1-n$ actions a; we will denote this sequence of actions by α_3.

As a result, we have two types of consistent histories:

1. histories in which the sequence of actions is of the type $[\alpha_1, a_0, a_0, \alpha_2^+, \alpha_3]$, and
2. histories in which the sequence of actions is of the type $[\alpha_1, a_0, \alpha_2^-, \alpha_3]$.

Strictly speaking, in addition to these two types of histories, we can have histories in which additional "dummy" actions are performed which do not change any states; however, since we are only interested in *minimal* histories, we can restrict ourselves only to histories of these two types.

Let us show that the given (empty) plan is successful if and only if the given quantified propositional formula is true.

The plan is successful if g is true for every possible (i.e., consistent and minimal) history. In the initial state, g is true. In the histories of the first type, we apply an action a_0 which makes g false. In the histories of the second type, we do not apply the action a_0 for the second time, so g remains true. Therefore, the empty plan guarantees that the goal g is satisfied if and only if all possible histories are of second type. In other words, the given (empty) plan is successful if and only if every history of first type – i.e., of the type $[\alpha_1, a_0, a_0, \alpha_2^+, \alpha_3]$ – is not minimal. In other words, the sufficient and necessary condition for a given plan to be successful is that every consistent history of the first type has a sub-sequence which forms a consistent history of the second type.

If the quantified formula is true, this means that for every consistent history h of the first type, for the sequence $x_1, \ldots, x_m$ which corresponds to α_1, there exist values $x_{m+1}, \ldots, x_n$ for which F is true. Then, we can do the following:

- delete from h the second a_0,
- select, from each pair of two consecutive actions a, a^- from α_2^+, a single action leading to the corresponding value x_i $(m+1 \le i \le n)$,

and thus get a sub-history h' of the second type. For this sub-history h', since F is true for these values, both x and t_{N+2} hold at the current state, and thus, this sub-history is consistent. Thus, if the quantified formula is true, no consistent history of first type is minimal and hence, the given plan is successful.

Vice versa, let us assume that the plan is successful. For every possible sequence $x_1, \ldots, x_m$, for the corresponding sequence α_1, we can add two actions a_0, $\alpha_2^+ = [a, a^-, a, a^-, \ldots, a, a^-]$ and $\alpha_3 = [a, a, \ldots, a]$, and get a consistent history h of the first type. Since the given plan is successful, this history cannot be minimal, i.e., it must have a consistent sub-history h' of the second type. In both types, we have exactly m actions α_1 before a_0, so every sub-history h' of h should have the same sequence α_1 as h itself. Hence, after this first stage, the fluents $f_1, \ldots, f_m$ representing variables $x_1, \ldots, x_m$ get the same values in h' as in h. The fact that h' is consistent means that x is true at the end, i.e., that $F(x_1, \ldots, x_m, \ldots, x_n)$ is true for the corresponding values $x_{m+1}, \ldots, x_n$. Thus, if every consistent history of the first type has a consistent sub-history of the second type, then for every $x_1, \ldots, x_m$, there exist values $x_{m+1}, \ldots, x_n$ for which F is true, i.e., the desired quantified formula is true.

Thus, the successfulness of the plan is indeed equivalent to the validity of the quantified formula. The theorem is proven.

Comment. In this proof, we used 3 actions: a, a^-, and a_0. Instead, we can use only two actions if, on the first and third stages, instead of using a single action to make a transition from one stage to another, we use a sequence of N identical actions, so that we are initially in some state q_1 (where a new fluent q_1 is true and all other new fluents $q_2, \ldots, q_N$ are false), and then the first $N-1$ applications of the action replace the state from q_i to q_{i+1} and change nothing else, and finally, the N-th application does the desired transformation and simultaneously makes q_1 true again. Then, on the stage t_{m+1}, instead of using a_0 one or twice, we can use either a, or $[a, a^-]$. In this case, we can also get the desired equivalence.

If we have only one possible action a, then all possible histories are simply sequences $[a, \ldots, a]$ of different length, so it is feasible to enumerate all possible subsequences and thus, feasible to check whether a given history is minimal. Thus, if we have a complete information about the initial state, it is feasible to check whether a given plan is successful. For the case of incomplete information, a plan is successful if it succeeds for all current states resulting from consistent histories; thus, plan checking is from the class $\Pi_1\mathbf{P}$=**coNP**. We can emulate the computation of a propositional formula and thus, for every propositional formula $F(x_1, \ldots, x_n)$, find a plan checking problem for which the plan is successful if and only if $\forall x_1 \ldots \forall x_n F(x_1, \ldots, x_n)$ is true. Hence, this problem is indeed **coNP**-complete.

Proof of Theorems 3 and 4. Let us first show that the corresponding planning problem indeed belongs to the desired class. The existence of a plan means that there exists a plan u_1 such that for every possible history u_2, *either* the history u_2 is consistent with our knowledge and the plan u_1 succeeds on the current state corresponding to u_2 (we will denote this by $S(u_1, u_2)$); *or* the history u_2 is not minimal, i.e., there exists a different history u_3 for which the sequence of

actions is a subsequence of the sequence of actions corresponding to u_2, and u_3 is also consistent with our knowledge (we will denote this property by $M(u_2, u_3)$).

Both binary predicates $S(u_1, u_2)$ and $M(u_2, u_3)$ are feasible to check. Therefore, the existence of a plan is equivalent to a formula $\exists u_1 \forall u_2 (S(u_1, u_3) \vee \exists u_3 M(u_2, u_3))$ with feasible predicates S and M, i.e., to the formula $\exists u_1 \forall u_2 \exists u_3 (S(u_1, u_3) \vee M(u_2, u_3))$ of the desired type.

The fact that the planning problem is complete in this class can be shown by a reduction to a propositional formula, a reduction which is similar to the one from the proof of Theorem 2; the only difference is that in addition to the above reduction – which, crudely speaking, simulates, during the period between the initial and the current state, the computation of the propositional expression corresponding to $M(u_2, u_3)$ – we must also, after the current state, simulate the computation of the expression corresponding to the formula $S(u_1, u_3)$.

Proof of Theorem 5. Let us first show that the corresponding planning problem belongs to the class **PSPACE**. Indeed, the existence of a plan means that there exists an action u_1 such that for every possible sensing result (if any) u_2 of this action, there exists a second action u_2, etc., such that for every history h_1 which is consistent with our initial knowledge and with the follow-up measurements, we either we get success, or there exists a "sub"-history h_2. Both success and "sub-history"-ness are feasible to check; thus, the existence of a plan is equivalent to a formula of the type $\exists u_1 \forall u_2 \ldots$, i.e., to the formula from the class **PSPACE**.

As we have shown in [3], this problem is **PSPACE**-complete even for $\mathcal{A}$, i.e., when no history is allowed. Thus, a more general problem from this class **PSPACE** should also be **PSPACE**-hard.

Proof of Theorem 6. Let us first show that the corresponding planning problem belongs to the class $\Pi_2\mathbf{P}$. Since we have unlimited sensing abilities, we do not change our planning abilities if, before we start any planning actions, we first sense the values of all the fluents. We may waste some time on unnecessary sensing, but the total execution time of a plan remains feasible if it was originally feasible; therefore, the existence of a feasible plan is equivalent to the existence of a feasible plan which starts with full sensing. The existence of such a plan means that for every consistent history u_1, either there is a plan u_2 which succeeds for the current state corresponding to u_1, or there exists a sub-history u_3 which is also consistent (which makes u_1 impossible). Checking whether a given plan succeeds for a given history is feasible, and checking whether u_3 is a consistent sub-history is also feasible, so the existence of a plan is equivalent to the formula $\forall u_1 (\exists u_2 P_1(u_1, u_2) \vee \exists u_3 P_2(u_1, u_3))$ for some feasible predicates P_1 and P_2. This formulas can be reformulated as $\forall u_1 \exists u_2 P(u_1, u_2)$ with $P(u_1, u_2)$ denoting $P_1(u_1, u_2) \vee P_2(u_1, u_2)$. Therefore, the problem belongs to the class $\Pi_2\mathbf{P}$.

As we have shown in [3], this problem is $\Pi_2\mathbf{P}$-complete even for $\mathcal{A}$, i.e., when no history is allowed. Thus, a more general problem from this class $\Pi_2\mathbf{P}$ should also be $\Pi_2\mathbf{P}$-hard.

Acknowledgments. This work was supported in part by NASA under cooperative agreement NCC5-209, by NSF grants No. DUE-9750858 and CDA-9522207, by United Space Alliance, grant No. NAS 9-20000 (PWO C0C67713A6), by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-95-1-0518, and by the National Security Agency under Grant No. MDA904-98-1-0561.

We are thankful to Luc Longpré and Tran Cao Son for valuable discussions.

References

1. Baral, C., Gabaldon, A., Provetti, A.: Formalizing Narratives Using Nested Circumscription, AI Journal **104** (1998) 107–164.
2. Baral, C., Gelfond, M., Provetti, A.: Representing Actions: Laws, Observations and Hypotheses, J. of Logic Programming **31** (1997) 201–243.
3. Baral, C., Kreinovich, V., Trejo, R.: Computational Complexity of Planning and Approximate Planning in Presence of Incompleteness, Proc. IJCAI'99 **2** 948–953 (full paper to appear in AI Journal).
4. Bylander, T.: The Computational Complexity of Propositional STRIPS Planning, AI Journal **69** (1994) 161–204.
5. Cai, J.-Y. et al.: The Boolean Hierarchy I: Structural Properties, SIAM J. on Computing **17** (1988) 1232–1252; Part II: Applications, in **18** (1989) 95–111.
6. Erol, K., Nau, D., Subrahmanian, V. S.: Complexity, Decidability and Undecidability Results for Domain-Independent Planning, AI Journal **76** (1995) 75–88.
7. Gelfond, M., Lifschitz, V.: Representing Actions and Change by Logic Programs, J. of Logic Programming **17** (1993) 301–323.
8. Gottlob, G., Scarcello, F., Sideri, M.: Fixed-Parameter Complexity in AI and Nonmonotonic Reasoning, Proc. LPNMR'99, Springer-Verlag LNAI **1730** (1999) 1–18.
9. Liberatore, P.: The Complexity of the Language $\mathcal{A}$, Electronic Transactions on Artificial Intelligence **1** (1997) 13–28.
10. Littman, M.: Probabilistic Propositional Planning: Representations and Complexity, Proc. AAAI'97 (1997) 748–754.
11. Papadimitriou, C.: Computational Complexity, Addison-Wesley, Reading, MA, 1994.
12. Papadimitriou, C., Wolfe, D.: The complexity of facets resolved, Proc. FOCS'85 (1985) 74–78; also, J. Computer and Systems Sciences **37** (1987) 2–13.
13. Papadimitriou, C., Yannakakis, M.: The complexity of facets (and some facets of complexity), Proc. STOC'82 (1982) 229–234; also, J. Computer and Systems Sciences **34** (1984) 244–259.
14. Rintanen, J.: Constructing Conditional Plans by a Theorem Prover, Journal of AI Research **10** (1999) 323–352.