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

Chitta Baral[1], Le-Chi Tuan[1], Raúl 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 problems, we can often complement the incompleteness of our *explicit* knowledge about the current state by using the *implicit* knowledge about this state which is contained in the description of the system's *past behavior*. For example, the information about the system's past failures is very important in planning diagnostic and repair. To describe planning which can use the information about the past, 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) to this more general class of planning problems.

1 Introduction

1.1 Planning problems: towards a more realistic formulation

Planning problems: traditional approach, with complete information about the initial state. Planning is one of the most important AI problems. Traditional AI formulations of this problem mainly cover 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, e.g., by the language $\mathcal{A}$ which was proposed in [8].

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., $\{\}$ or $\{f_1, f_3\}$. We are assuming that we have a complete knowledge about the initial state: e.g., $\{f_1, f_3\}$ means that in the initial state, properties f_1 and f_3 are true, while all the other properties $f_2, f_4, \ldots$ are false. The properties of the initial state are described by formulas of the type "initially F," where F is a *fluent literal*, i.e., either a fluent f_i or its negation $\neg f_i$.

There is also a finite set $\mathcal{A}$ of possible *actions*. At each moment of time, an agent can execute an action. 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. A reasonably straightforward semantics describes how the state changes after an action:

- If before the action a, the literals $F_1, \ldots, F_m$ were true, and the domain description contains a rule according to which a causes F if $F_1, \ldots, F_m$, then this rule is *activated*, and after the execution of action a, F becomes true. Thus, for some fluents f_i, we will conclude f_i and for some other, that $\neg f_i$ holds in the resulting state.
- If for some fluent f_i, no activated rule enables us to conclude that f_i is true or false, this means that the execution of action a does not change the truth of this fluent. Therefore, f_i is true in the resulting state if and only if it is true in the old state. (This case represents *inertia*.)

Formally, a *domain description* D is a finite set of *value propositions* of the type "initially f" (which describe the initial state), and a finite set of *effect propositions* of the type "a causes f if $f_1, \ldots, f_m$" (which describe results of actions). A *state* s is a finite set of fluents. The *initial state* s_0 consists of all the fluents f_i for which the corresponding value proposition "initially f_i" is contained in the domain description. (Here we are assuming that we have complete information about the initial situation.) We say that a fluent f_i *holds* in s if $f_i \in s$; otherwise, we say that $\neg f_i$ holds in s.

The *transition function* $res(a, s)$ which describes the effect of an action a on a state s is defined as follows:

- we say that an effect proposition "a causes F if $F_1, \ldots, F_m$" is *activated* in a state s if all m fluent literals $F_1, \ldots, F_n$ hold in s;
- we define $V_D^+(a, s)$ as the set of all fluents f_i for which a rule "a causes f_i if $F_1, \ldots, F_m$" is activated in s;
- similarly, we define $V_D^-(A, S)$ as the set of all fluents f_i for which a rule "a causes $\neg f_i$ if $F_1, \ldots, F_m$" is activated in s;
- if $V_D^+(a, s) \cap V_D^-(a, s) \neq \emptyset$, we say that the result of the action a is *undefined*;
- if the result of the action a is not undefined in a state s (i.e., if $V_D^+(a, s) \cap V_D^-(a, s) = \emptyset$), we define $res(a, s) = (s \cup V_D^+(a, s)) \setminus V_D^-(a, s)$.

A *plan* α is a sequence of of actions $\alpha = [a_1, \ldots, a_n]$; the result $res(a_n, res(a_{n-1}, \ldots, res(a_1, s) \ldots))$ of applying these actions to the state s is denoted by $res(\alpha, s)$.

To complete the description of deterministic planning, we must formulate possible *objectives*. In general, as an objective, we can take a complex combination of elementary properties (fluents) which characterize the final state; for example, a typical objective of an assembling manufacture robot is to reach the state of the world in which all manufactured items are fully assembled. To simplify the description of the problem, we can always add this combination as a new fluent; thus, without losing generality, it is sufficient to consider only objectives of the type $f \in \mathcal{F}$.

In these terms, the *planning* problem can be formulated as follows: given a set of fluents $\mathcal{F}$, a goal $f \in \mathcal{F}$, a set of actions $\mathcal{A}$ and a set of rules D describing how these actions affect the state of the world, to find a sequence of actions $\alpha = [a_1, \ldots, a_k]$ that, when executed from the initial state of the world s_0, makes f true. The problem of *plan checking* is, given $\mathcal{F}$, $\mathcal{A}$, a goal, and a sequence of actions α, to check whether the goal becomes true after execution of α in the initial state.

Next step: planning in case of incomplete information about the initial state. The language $\mathcal{A}$ describes allows planning in the situations with *complete* information, when we know exactly which fluents hold in the initial state and which don't. In real life, we often have only *partial* information about the initial state: about some fluents, we know that they are true in the initial state, about some other fluents, we know that they are false in the initial state; and it is also possible that about some fluents, we do not know whether they are initially true or false.

For example, when we want a mobile robot to reach a certain point, we often do not have a complete information about the state of the world; this is especially true in space applications, when the goal of the robot is to explore new environments whose state is initially unknown. When we plan a diagnostic and repair of a complex object, be it a computer, a car, etc., we do not know which parts are functioning correctly and which parts are not – this is exactly what we are trying to find out. In terms of fluents, this means that we do not know the initial values of the fluents which describe the correct functionality of the system's parts.

Such situations can also be easily described by a simple modification of the above language $\mathcal{A}$. Namely, if for some fluent f, neither the statement "$\mathsf{initially}$ f", not the statement "$\mathsf{initially}$ $\neg f$" are given, we assume that two different initial situations are possible: when f if initially true, and when $\neg f$ is false in the initial state. As a result, instead of a single initial state s_0, we may have several different initial states which are consistent with our knowledge about the system.

In this case, the notion of a successful plan becomes slightly more complex: namely, 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 $res(\alpha, s)$.

Adding sensing actions. In real-life planning problems like the above-mentioned problems of robotic motion or system diagnostic, a reasonable plan involves using *sensors* to find the missing information. Even in simple real-life planning situations, it is often necessary to determine the missing information. For example, if we want the door closed, the required action depends on whether the door was initially open (then we close it), or it was already closed (then we do nothing). Therefore, if we do not know whether the door was initially closed or not, we better somehow find it out, and then, depending on the result of this investigation, perform the corresponding action.

To describe such activities, we must include *sensing actions* – e.g., an action $check_i$ which checks whether the fluent f_i holds in a given state – to our list of actions, and allow *conditional* plans, i.e., plans in which the next action depends on the result of the previous sensing action.

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.

Since we will now be dealing with incompleteness of information about the real world, we will need to reason with the agent's knowledge about the world. A *k-state* is defined as pair $\langle s, \Sigma\rangle$, where s is the *actual* state, and Σ is the set of all possible states where the agent thinks it may be in. Initially, the set Σ_0 consists of all the states s for which:

– a fluent f_i is true ($f_i \in s$) if the domain description D contains the proposition "initially f_i";
– a fluent f_i is false ($f_i \notin s$) if the domain description D contains the proposition "initially $\neg f_i$".

If neither the proposition "initially f_i", nor the proposition "initially $\neg f_i$" are in the domain description, then Σ_0 contains some states with f_i true *and* others with f_i false. The actual initial state s_0 can be any state from the set Σ_0. The transition function due to action execution is defined as follows:

– for proper (*non-sensing*) actions, $\langle s, \Sigma\rangle$ is mapped into $\langle res(a, s), res(a, \Sigma)\rangle$, where:
  • $res(a, s)$ is defined as in the case of complete information, and
  • $res(a, \Sigma) = \{res(a, s') \mid s' \in \Sigma\}$.
– for a *sensing* action a which senses fluents $f_1, \ldots, f_k$ – i.e., for which sensing propositions "a determines f_i" belong to the domain D – the actual state s remains unchanged while Σ is down to only those states which have the same values of f_i as s: $\langle s, \Sigma\rangle \to \langle s, \Sigma'\rangle$, where

$$\Sigma' = \{s' \in \Sigma \mid \forall i\, (1 \le i \le k \to (f_i \in s' \leftrightarrow f_i \in s))\}$$

In the presence of sensing, an action plan may no longer be a pre-determined sequence of actions: if one of these actions is sensing, then the next action may depend on the result of that sensing. In general, the choice of a next action may depend on the results of all previous sensing actions. Such an action plan is called a *conditional plan.*

Possibility of knowledge about the past. In the situations when we only have a partial information about the current (present) state, the additional information can be deduced from knowing the *history* of the system's behavior. This additional information about the past is extremely important in diagnostic problems: if we know what types of faulty behavior the system exhibited in the past, it helps in diagnostics (sometimes this information about the past is even sufficient for a successful repair, and no additional sensing is necessary). Similarly, when a medical doctor plans a cure, information about past diseases is as important (and sometimes even more important) than the results of different tests ("sensing actions").

Since this additional information is very important in many practical planning problems, it is desirable to include this information into the corresponding AI formalisms.

To describe the use of knowledge about the past in 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 allows:

- 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 $t(s_i) \leq m$ (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.

Let us give an example of such a situation. If a lamp is not broken, then, when we switch it on, the light bulb should be switched on. If in the past, we applied the action *turn_on* but the lamp did not go on, this means that the lamp was broken at that time, and, if we know of no repair actions performed in the past, we can therefore conclude that the lamp is still broken. This narrative can be described by the following rules: "*switch_on* causes *lamp_on* if $\neg$*broken*", "*switch_on* occurs_at s_1", "s_1 precedes s_2", "$\neg$*lamp_on* at s_2". From these rules, we can conclude that the lamp is currently *broken*.

1.2 Computational complexity of planning problem: why it is important, what is known, and what we are planning to do

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. As we have mentioned earlier, such situations are described, e.g., by the language $\mathcal{A}$ which was proposed in [8]. The complexity of planning in $\mathcal{A}$ was analyzed in our earlier paper [3].

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: $t_{\mathcal{U}}(x) \le p(|w|)$ (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 $m \le p(|D|)$) 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 $m \le P_0$, 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 $\le P_0 - 1$ steps. Since a plan of length $\le P_0$ 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 $\le P_0$ by applying the algorithm which checks the existence of the plan $\le P_0 = p(|D|)$ 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 [5, 7, 11]. For the language $\mathcal{A}$, computational complexity of planning was first studied in [10]; the results about the computational complexity of different planning problems in $\mathcal{A}$ are overviewed in [3, 15].

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].

What we are planning to do. We have mentioned that a more realistic description of a planning problem involves the use of history (information about the past) in planning. 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 classify these problems to different levels of the polynomial hierarchy. For precise definitions of the polynomial hierarchy, see, e.g., [12]. 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., [12]) 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 nor static causal laws are allowed.

2.1 Complexity of plan checking

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 $\Pi_2\mathbf{P}$-complete.*
- *For situations with incomplete information about the initial state, the plan checking problem is $\Pi_2\mathbf{P}$-complete.*

Comment. The problem remains $\Pi_2\mathbf{P}$-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.

2.2 Complexity of planning

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., [6, 12]). This hierarchy started with the discovery of the first such class – the class **DP** [13, 14]. 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 $S_1 = \{w \mid \exists u_1 P_2(u_1, w)\}$ from the class $\mathbf{NP}$ and a set $S_2 = \{w \mid \forall u_2 P_1(u_2, w)\}$ from the class $\mathbf{coNP}$, 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 $\mathbf{NP}$. Such sets represent the difference class $\mathbf{DP}$, 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 $P(u_1, \ldots, u_k, w)$ is equal to the result $B(P_1, \ldots, P_m)$ of substituting, into the Boolean expression $B(v_1^{j_1}, \ldots, v_m^{j_m})$, instead of each variable $v_i^{j_i}$, a feasible predicate P_i which 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 [9], 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.

What do these complexity results mean in practical terms? At first glance, they may sound gloomy: even **NP**-complete problems are extremely difficult to solve, and the most realistic formulations of the planning problem (with sensing) lead to **PSPACE**-complete problems, i.e., problems at the high end of the polynomial hierarchy. However, they do not sound so gloomy if we take into consideration that these results are about the *worst-case* complexity, and the high worst-case complexity of the problem does not mean that we cannot have good algorithm for many (or even for most) practical instances of this problem.

In plain words, no matter how good a feasible planning algorithm may be, there will always be cases when this algorithm will fail. Our goal is therefore, to design feasible algorithms which will succeed on as many practical planning problems as possible.

Even the traditional planning problem, with no sensing and complete information about the initial state, is known to be **NP**-hard; this complexity result does not prevent us from having successful planners which help in solving many practical planning problems. For situations with incomplete information about the initial state, several ideas of approximate planning were proposed in [4]; the corresponding simplified algorithms are much faster than the algorithms for solving the original planning problem (and the complexity of the corresponding approximate planning problem is indeed smaller; see, e.g., [3]) – the downside being, of course, that sometimes, these approximate algorithms fail to find a plan.

It is desirable to extend these (and other) heuristic planning algorithms to situations when some information about the current state comes in the form of the knowledge about the system's past behavior.

3 Proofs

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

Proof of Theorem 2: main idea. 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 $\Pi_2\mathbf{P}$.

To complete the proof, we must prove that the plan checking problem is $\Pi_2\mathbf{P}$-complete. To show it, we prove that the known $\Pi_2\mathbf{P}$-complete problem – namely, the problem of checking, for a given propositional formula F, whether a formula $\forall x_1 \ldots \forall x_m \exists x_{m+1} \ldots \exists x_n\, F(x_1, \ldots, x_n)$ is true – can be reduced to plan checking. It is sufficient to 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. This reduction is done similarly to the proofs from [3] (a detailed proof is posted at http://www.cs.utep.edu/vladik/2000/tr00-13.ps.gz).

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 proofs from [3] and 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, 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 a 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 formula 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, to Tran Cao Son, and to the anonymous referees 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. Baral, C., Son, T.: Approximate reasoning about actions in presence of sensing and incomplete information, Proc. ILPS'97 387–401.
5. Bylander, T.: The Computational Complexity of Propositional STRIPS Planning, AI Journal **69** (1994) 161–204.
6. 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.
7. Erol, K., Nau, D., Subrahmanian, V. S.: Complexity, Decidability and Undecidability Results for Domain-Independent Planning, AI Journal **76** (1995) 75–88.
8. Gelfond, M., Lifschitz, V.: Representing Actions and Change by Logic Programs, J. of Logic Programming **17** (1993) 301–323.
9. Gottlob, G., Scarcello, F., Sideri, M.: Fixed-Parameter Complexity in AI and Nonmonotonic Reasoning, Proc. LPNMR'99, Springer-Verlag LNAI **1730** (1999) 1–18.
10. Liberatore, P.: The Complexity of the Language $\mathcal{A}$, Electronic Transactions on Artificial Intelligence **1** (1997) 13–28.
11. Littman, M.: Probabilistic Propositional Planning: Representations and Complexity, Proc. AAAI'97 (1997) 748–754.
12. Papadimitriou, C.: Computational Complexity, Addison-Wesley, Reading, MA, 1994.
13. 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.
14. 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.
15. Rintanen, J.: Constructing Conditional Plans by a Theorem Prover, Journal of AI Research **10** (1999) 323–352.