The Complexity of Agent Design Problems: Determinism and History Dependence

MichaelWooldridge and Paul E.Dunne
Dept. of Computer Science, University of Liverpool, Liverpool L69 7ZF, United Kingdom

Abstract. The *agent design* problem is as follows: given a specification of an environment, together with a specification of a task, is it possible to construct an agent that can be guaranteed to successfully accomplish the task in the environment? In this article, we study the computational complexity of the agent design problem for tasks that are of the form "achieve this state of affairs" or "maintain this state of affairs". We consider three general formulations of these problems (in both non-deterministic and deterministic environments) that differ in the nature of what is viewed as an "acceptable" solution: in the least restrictive formulation, no limit is placed on the number of actions an agent is allowed to perform in attempting to meet the requirements of its specified task. We show that the resulting decision problems are intractable, in the sense that these are non-recursive (but recursively enumerable) for achievement tasks, and non-recursively enumerable for maintenance tasks. In the second formulation, the decision problem addresses the existence of agents that have satisfied their specified task within some given number of actions. Even in this more restrictive setting the resulting decision problems are either PSPACE-complete or NP-complete. Our final formulation requires the environment to be *history independent* and bounded. In these cases polynomial time algorithms exist: for deterministic environments the decision problems are NL-complete; in non-deterministic environments, P-complete.

1. Introduction

We are interested in building agents that can autonomously act to accomplish tasks on our behalf in complex, unpredictable environments. Other researchers with similar goals have developed a range of software architectures for agents [17]. In this article, however, we focus on the underlying decision problems associated with the deployment of such agents. Specifically, we study the *agent design* problem [15]. This problem may be informally stated as follows:

> Given a specification of an environment, together with a specification of a task that we desire to be carried out on our behalf in this environment, is it possible to construct an agent that can be guaranteed to successfully accomplish the task in the environment?

The *type* of task to be carried out is crucial to the study of this problem. In this article, we address ourselves to the two most common types of tasks encountered in artificial intelligence: *achievement tasks*, (where an agent is required to achieve some state of affairs), and *maintenance tasks*, (where an agent is required to maintain some state of affairs).

In [15], it was proved that in the most general case, the agent design problem for both achievement and maintenance tasks is PSPACE-complete. In this research note, we extend the results of [15] considerably. We begin by formally defining what we mean by agents and environments, and introduce achievement and maintenance design tasks. We then consider a number of different requirements for what constitutes a "successful" agent: informally these requirements pertain to whether or not some limit is placed on the total number of actions that an agent is allowed to perform in order to accomplish its task. Achievement and maintenance tasks turn out to be equivalent via LOGSPACE reductions *except* in the setting when no upper limit is placed on the number of actions allowed and *unbounded* environments may occur as instances. We show in section 3.1, that unless "successful" agents are compelled to complete their tasks within some number of steps, the agent design problems are *provably intractable* (i.e., intractable irrespective of assumptions regarding classical open questions in computational complexity theory, e.g. P=?NP, P=?PSPACE etc.)

In section 3.2 we focus on *finite* agent design problems, whereby agents must achieve some state of affairs within at most some number of steps (finite achievement tasks) or maintain a state of affairs for at least some number of steps (finite maintenance tasks) In particular, we identify the following environmental characteristics that can affect the complexity of the (finite) agent design problem:

– *determinism* refers to whether or not the next state of the environment is uniquely defined, given the history of the system to date and the agent's current action;

– *history dependence* refers to whether or not the next state of the environment is simply a function of the agent's current action and the environment's current state, or whether previous environment states and actions can play a part in determining the next state.

In section 4, we discuss related work and present some conclusions. Throughout the article, we assume some familiarity with complexity theory [11].

2. Agents, Environments, and Runs

The systems of interest to us consist of an agent situated in some particular environment. The agent interacts with the environment by performing actions upon it, and the environment responds to these actions with changes in state. It is assumed that the environment may be in any of a finite set $E = \{e_0, e_1, \ldots, e_n\}$ of instantaneous states. Agents are assumed to have a repertoire of possible actions available to them, which transform the state of the environment. Let $Ac = \{\alpha_0, \alpha_1, \ldots, \alpha_k\}$ be the (finite) set of actions. The behaviour of an environment is defined by a *state transformer function*, τ. This function allows for non-determinism in the environment: a number of possible environment states can result from performing an action in apparently the same circumstances. However, in our case τ is not simply a function from environment states and actions to sets of environment states, but from *runs* and actions to sets of environment states. This allows for the behaviour of the environment to be dependent on the *history* of the system — the previous states of the environment and previous actions of the agent can play a part in determining how the environment behaves.

To avoid excessive repetition the following notational conventions are used.

For H any set defined by a relationship of the form $H = \cup_{i=0}^{\infty} H^{(i)}$, where $H^{(i)} \cap H^{(j)} = \emptyset$ whenever $i \neq j$, $H^{(\leq k)}$ denotes the subset $\cup_{i=0}^{k} H^{(i)}$ of H.

Where $t = t_1 t_2 \cdots t_k \cdots t_n$ is a finite sequence, $last(t)$ will denote the last element, t_n.

DEFINITION 1. *An* environment *is a quadruple* $Env = \langle E, e_0, Ac, \tau \rangle$, *where* E *is a finite set of* environment states *with* e_0 *distinguished as the* initial state*; and* Ac *is a finite set of* available actions. *Let* S_{Env} *denote all sequences of the form:*

$$e_0 \cdot \alpha_0 \cdot e_1 \cdot \alpha_1 \cdot e_2 \cdots$$

where e_0 *is the initial state, and* $\forall i,\ e_i \in E,\ \alpha_i \in Ac$.

Let S^E *be the subset of such sequences that end with a state. The* state transformer function τ *is a total mapping*

$$\tau : S^E \times Ac \to \wp(E)$$

We wish to focus on that subset of S^E *representing* possible *sequences of states and actions which could arise, i.e., if* $s \in S^E$ *and* $\tau(s, \alpha) = \emptyset$, *then certainly no sequence in* S^E *having* $s \cdot \alpha$ *as a prefix can result from the initial state. In order to make this precise, we define the concept of a* run.

The subset of S^E *defining the* runs in the environment *Env, is*

$$R_{Env} = \bigcup_{k=0}^{\infty} R_{Env}^{(k)}$$

where, $R_{Env}^{(0)} = \{e_0\}$, *and*

$$R_{Env}^{(k)} = \bigcup_{r \in R_{Env}^{(k-1)}} \ \bigcup_{\{\alpha \in Ac \mid \tau(r,\alpha) \neq \emptyset\}} \{r \cdot \alpha \cdot e \mid e \in \tau(r, \alpha)\} \text{ if } k > 0$$

Thus, $R_{Env}^{(k)}$, is the set of histories that could result after exactly k actions from the initial state e_0. Similarly, $R_{Env}^{(\leq k)}$ indicates the set of histories that could result after at most *k actions from the initial state.*

An environment is k-bounded *if $R_{Env}^{(k+1)} = \emptyset$; it is* bounded *if k-bounded for some finite value k, and* unbounded *otherwise.*

We denote by R^{Ac} the set $\{r \cdot \alpha \mid r \in R_{Env}\}$ so that, with a slight abuse of notation, we subsequently interpret τ as a total mapping, $\tau : R^{Ac} \to \wp(E)$.

A run, r, has terminated *if for all actions α in Ac, $\tau(r \cdot \alpha) = \emptyset$, thus if $r \in R_{Env}^{(k)}$ has terminated then r does not form the prefix of any run in $R_{Env}^{(k+1)}$. The subset of $R_{Env}^{(k)}$ comprising terminated runs is denoted $T_{Env}^{(k)}$, with T_{Env} the set of all terminated runs. We note that if Env is k-bounded then all runs in $R_{Env}^{(k)}$ have terminated.*

The length *of a run, $r \in R_{Env}$ is the total number of actions and states occurring in r and is denoted by $\mid r \mid$. In a k-bounded environment, all runs have length at most $2k+1$ i.e., $k+1$ states and k actions.*

It should be noted that for any environment, *Env*, and run r in $R_{Env}^{(k)}$: r has either *terminated* (that is for all actions, α, $\tau(r \cdot \alpha) = \emptyset$) or there is at least one action α for which $r \cdot \alpha$ is a (proper) prefix of some non-empty set of runs in $R_{Env}^{(k+1)}$.

The state transformer function, τ, is represented by a deterministic Turing machine program description T_τ with the following characteristics: T_τ takes as its input a run $r \in R^{Ac}$) and a state $e \in E$. If T_τ accepts within $|r|$ steps then $e \in \tau(r)$; if T_τ has not accepted its input within this number of steps then $e \notin \tau(r)$.[1]

It is also worth commenting on the notion of history dependence. Many environments have this property. For example, consider the travelling salesman problem [11, p13]: history dependence arises because the salesman is not allowed to visit the same city twice. Note that it is possible to transform a history dependent *bounded* environment into a history independent one, by encoding information about prior history into an environment state. However, this can only be done at the expense of an increase in the number of environment states. Intuitively, given a history dependent environment with state set E, which has an associated set of runs R, we would need to create $|R \times E|$ environment states. Since $|R|$ could be exponential in the size of $Ac \times E$, this could lead to an exponential blow-up in the size of the state space.

[1] The requirement that T_τ reach a decision within $|r|$ moves is *not* as restrictive as it appears: we may admit programs, T_τ, that require $|r|^k$ steps to decide $e \in \tau(r)$, by constructing a new environment, $Pad(Env)$ from *Env*, in which runs $r \in R_{Env}$ are "embedded" in runs $\pi(r)$ of length $O(|r|^k)$ in $R_{Pad(Env)}$, in such a way that $\tau(r) \subseteq \tau_{Pad}(\pi(r))$ and $\tau_{Pad}(s)$ is computable in $|s|\ |E_{Pad}|$ steps.

We view *agents* as performing actions upon the environment, thus causing the state of the environment to change. In general, an agent will be attempting to "control" the environment in some way, in order to carry out some task on our behalf. However, as environments may be non-deterministic, an agent generally has only partial control over its environment.

DEFINITION 2. *An* agent*, Ag, in an environment* $Env = \langle E, e_0, Ac, \tau \rangle$ *is a mapping*

$$Ag : R_{Env} \;\rightarrow\; Ac \cup \{\otimes\}$$

The symbol $\otimes$ *is used to indicate that the agent has finished its operation: an agent may only invoke this on terminated runs,* $r \in T_{Env}$*, an event that is referred to as the agent having* no allowable actions.

We stress that our definition of agent does *not* allow the termination action, $\otimes$ to be invoked on r if there is any allowable action α defined for r. While this may seem unduly restrictive it reflects the fact that agents may not choose to halt "arbitrarily". It may be noted that an agent, Ag, having no allowable action for a run r in $R(Ag, Env)$ is equivalent to r being terminated, i.e., in $T(Ag, Env) \subseteq T_{Env}$.

In general, the state transformer function τ (with domain R^{Ac}) is *non-deterministic*: for a given, r, $\tau(r)$ describes a set of *possible* next states for the environment. The interpretation is that *exactly one* of these states will result, but which one is unpredictable. In contrast, our view of agents is *deterministic*: for a given $r \in R_{Env}$, the agent prescribes exactly one action with which to *continue* a run.

DEFINITION 3. *A* system*, Sys, is a pair* $\langle Env, Ag \rangle$ *comprising an environment and an agent operating in that environment. A sequence* $s \in R_{Env} \cup R^{Ac}$ *is called a* possible run of the agent Ag in the environment Env *if*

$$s = e_0 \cdot \alpha_0 \cdot e_1 \cdot \alpha_1 \cdots$$

satisfies

1. e_0 *is the initial state of E, and* $\alpha_0 = Ag(e_0)$*;*

2. $\forall k > 0$,

$$\begin{array}{rcl} e_k & \in & \tau(e_0 \cdot \alpha_0 \cdot e_1 \cdot \alpha_1 \cdots \alpha_{k-1}) \text{ where} \\ \alpha_k & = & Ag(e_0 \cdot \alpha_0 \cdot e_1 \cdot \alpha_1 \cdots e_k) \end{array}$$

Thus, with τ being non-deterministic, an agent may have a number of different possible runs in any given environment.

We use $R(Ag, Env)$ to denote the set of runs in R_{Env} that are possible runs of an agent Ag in the environment Env, and $T(Ag, Env)$ the subset of $R(Ag, Env)$ that are terminated runs, i.e., belong to T_{Env}.

3. Tasks for Agents

We build agents in order to carry out *tasks* for us. The task to be carried out must be *specified* by us. A discussion on the various ways in which we might specify tasks for agents appears in [15], but for the purposes of this article, we will be concerned with just two types of tasks:

1. *Achievement tasks*: are those of the form "achieve state of affairs φ".

2. *Maintenance tasks*: are those of the form "maintain state of affairs ψ".

Intuitively, an achievement task is specified by a number of *goal states*; the agent is required to bring about one of these goal states (we do not care which one). Achievement tasks are probably the most commonly studied form of task in artificial intelligence. Many well-known AI problems (including, e.g., the blocks world) are instances of achievement tasks.

Just as many tasks can be characterised as problems where an agent is required to bring about some state of affairs, so many others can be classified as problems where the agent is required to *avoid* some state of affairs. As an extreme example, consider a nuclear reactor agent, the purpose of which is to ensure that the reactor never enters a "meltdown" state. Somewhat more mundanely, we can imagine a software agent, one of the tasks of which is to ensure that a particular file is never simultaneously open for both reading and writing. We refer to such tasks as *maintenance* tasks.

We wish to express questions regarding whether a suitable agent exists as decision problems. An instance of such a problem must certainly encode both the environment *Env* and the set of states to be achieved or avoided. In addition to these, however, we have to consider another issue concerning what constitutes a *"successful"* agent.

We can consider, in the least "restrictive" setting, the question of whether an agent, Ag, exists whose every run eventually achieves (always avoids) some specified set of states. It should be noted that in a bounded environment, *every* run, r in R_{Env} forms the prefix of some (non-empty set of) *terminated* runs in T_{Env}. In the context of what forms an instance of the decision problem, if it is assumed that the environment is bounded then the actual bound, i.e., the least value k for which $R_{Env}^{(k+1)} = \emptyset$, does *not* have to be encoded as part of the instance.

So we can define decision problems for achievement and maintenance tasks as, DEFINITION 4. *The decision problem* ACHIEVEMENT AGENT DESIGN *(AD) takes as input a pair* $\langle Env, G\rangle$ *where* $G \subseteq E$ *is a set of environment states called the* Good *states.* $AD(\langle Env, G\rangle)$ *returns true if and only if there is an agent, Ag, and value* $n_{ag} \in \mathbb{N}$*, with which*

$$\forall r \in R^{\leq n_{ag}}(Ag, Env)\exists s \in R^{\leq n_{ag}}(Ag, Env)\exists g \in G \text{ such that } \begin{cases} r \text{ is a prefix of } s \\ \text{and} \\ g \text{ occurs in } s \end{cases}$$

This definition expresses the requirement that a "successful" agent must (eventually, hence within some finite number of actions – n_{ag}) reach some state in G on every run. In the form given, this decision problem applies both to unbounded and bounded environments, however, in the latter case, we can simplify the condition governing an agent's success to

$$\forall r \in T(Ag, Env)\exists g \in G \qquad g \text{ occurs in } r$$

That is, only *terminated* runs need be considered.

DEFINITION 5. *The decision problem* MAINTENANCE AGENT DESIGN *(MD) takes as input a pair* $\langle Env, B\rangle$ *where* $B \subseteq E$ *is a set of environment states called the* Bad *states.* $MD(\langle Env, B\rangle)$ *returns true if and only if, there is an agent, Ag, with which*

$$\forall r \in R(Ag, Env)\forall b \in B, \qquad b \textit{ does } \text{not } \textit{occur in } r$$

Thus, a successful agent must avoid any state in B on every run. As with the formulation of ACHIEVEMENT AGENT DESIGN, the decision problem applies to both unbounded and bounded environments, with

$$\forall r \in T(Ag, Env)\forall b \in B \qquad b \text{ does not occur in } r$$

the condition governing success in the latter case.

These phrasings of the decision problems AD and MD do not impose any constraints on the *length* of runs in $R(Ag, Env)$: of course, if Env is bounded then this length will be finite. It turns out, however, that significant computational difficulties arise from these formulations.

As a result we also consider a rather more restrictive notion in which success is defined with respect to runs in $R_{Env}^{(\leq k)}$. In decision problem formulations for this class we could choose to supply the value of k (in unary) as *part of the instance*. We prefer, however, to define decision problems in which only runs involving *at most* $|E \times Ac|$ actions are considered. We note that this formulation, while encompassing all $|E \times Ac|$-bounded environments, does not *require* instances to be so (or even to be bounded at all).

DEFINITION 6. *The decision problem* FINITE ACHIEVEMENT DESIGN *(FAD) takes as input a pair* $\langle Env, G\rangle$ *as for AD.* $FAD(\langle Env, G\rangle)$ *returns true if and only if there is an agent, Ag, with which,*

$$\forall r \in T^{(\leq|E\times Ac|-1)}(Ag, Env) \cup R^{(|E\times Ac|)}(Ag, Env) \exists g \in G \textit{such that } g \textit{ is in } r$$

DEFINITION 7. *The decision problem* FINITE MAINTENANCE DESIGN *(FMD) takes as input a pair* $\langle Env, B\rangle$ *as for MD.* $FMD(\langle Env, B\rangle)$ *returns true if and only if there is an agent Ag, with which,*

$$\forall r \in R^{(\leq|E\times Ac|)}(Ag, Env) \forall b \in B, b \textit{ does } \text{not} \textit{ occur in } r$$

We observe the following concerning the definitions of *FAD* and *FMD*. Firstly, although examining only agents with runs in $R_{Env}^{(\leq|E\times Ac|)}$ appears very restrictive, this limitation has no significant impact with respect to computational complexity: suppose we wished to admit agents with runs in $R_{Env}^{(\leq t)}$ for values $t = |E \times Ac|^k$, and defined variations FAD_k and FMD_k for such cases. Trivial reductions show that $FAD_k \leq_{\log} FAD$ and $FMD_k \leq_{\log} FMD$: given an instance $\langle Env, G\rangle$ of FAD_k, simply create an instance $\langle Env_k, G\rangle$ of *FAD* by "padding" the state set of *Env* with sufficiently many new states $\{n_1, \cdots, n_m\}$ none of which is reachable from the initial state. This reduction increases the size of an instance only polynomially.

We use the term *critical run of the environment*, to describe runs in the set

$$C_{Env} = T_{Env}^{(\leq|E\times Ac|-1)} \cup R_{Env}^{(|E\times Ac|)}$$

with $C(Ag, Env)$ – the *critical runs of an agent Ag* in *Env* being

$$C(Ag, Env) = T^{(\leq|E\times Ac|-1)}(Ag, Env) \cup R^{(|E\times Ac|)}(Ag, Env)$$

As a second point, we note that formulations other than quantifying by,

$$\forall r \in T^{(\leq|E\times Ac|-1)}(Ag, Env) \cup R^{(|E\times Ac|)}(Ag, Env)$$

could be used, e.g. requiring the agent to have terminated after at most $|E\times Ac|$ actions, considering only terminated runs in $R^{(|E\times Ac|)}(Ag, Env)$ but ignoring those runs which have yet to terminate. Neither of these alternatives yield significant changes in computational complexity of the specific design tasks we examine.

For the FINITE AGENT DESIGN problems, the definition of "success' employed corresponds to:

FINITE ACHIEVEMENT AGENT DESIGN: "is there an agent which always reaches some state in *G after at most* $|E \times Ac|$ actions?"

FINITE MAINTENANCE AGENT DESIGN: "is there an agent which manages to avoid every state in *B for at least* $|E \times Ac|$ actions?"

We now consider two variations of achievement and maintenance design problems, which depend on whether or not the environment is *deterministic* — that is, whether or not the next state of the environment is uniquely determined, given the previous history of the environment and the current action performed by the agent:

non-det. $\tau : R^{Ac} \to \wp(E)$, i.e., τ maps from runs to *sets* of possible states. This is the most general context introduced in Definition 1, in which τ is non-deterministic.

det. $\tau : R^{Ac} \to E \cup \{\emptyset\}$, i.e., τ maps from runs to *at most one* next state. Thus τ is deterministic: either there is no allowable continuation of a run, r, on a given action α (i.e., $\tau(r \cdot \alpha) = \emptyset$) or there is exactly one state into which the environment is transformed.

We use the notations AD_X (MD_X, FAD_X, FMD_X) where $X \in \{non\text{-}det, det\}$ to distinguish the different settings.

3.1. PROVABLE INTRACTABILITY OF *AD* AND *MD*

The results of this subsection indicate why we shall mainly consider the finite agent design variants. may be unbounded or are required to be bounded environments.

THEOREM 1. *Let*

$$L_{univ} = \{ \langle M, x \rangle \mid x \text{ is accepted by the Turing machine } M \}$$

where M is a deterministic single tape Turing machine using a binary alphabet. Then:

$$L_{univ} \leq_p AD_{det}$$

Proof: Let $\langle M, x \rangle$ be an instance of L_{univ}. We form an instance $\langle Env_{Mx}, G \rangle$ of AD_{det}, for which $\langle M, x \rangle$ is in L_{univ} if and only if $AD_{det}(\langle Env_{Mx}, G \rangle)$ returns true.

$$E_{Mx} = \{e_0, accept, reject\} \; ; \; Ac_{Mx} = \{\alpha\} \; ; \; G = \{accept\}$$

τ_{Mx} is defined as follows:

$$\tau_{Mx}(r \cdot \alpha) = \begin{cases} \emptyset & \text{if } \; last(r) \in \{accept, reject\} \\ accept & \text{if } \; r \in R^{(k)}_{Env_{Mx}} \text{ and } M \text{ accepts } x \text{ after exactly } k \text{ moves, } (k \geq 0) \\ reject & \text{if } \; r \in R^{(k)}_{Env_{Mx}} \text{ and } M \text{ rejects } x \text{ after exactly } k \text{ moves, } (k \geq 0) \\ e_0 & \quad \text{otherwise} \end{cases}$$

Noting that there is *at most one* run in $R^{(k)}_{Env_{Mx}}$ for any $k \geq 0$ and that this is of the form $e_0 \cdot (\alpha \cdot e_0)^k$, a Turing machine, $T_{\tau_{Mx}}$ program computing τ_{Mx} simply checks if its input run, r is in $R^{(k)}_{Env_{Mx}}$ and then simulates the first k moves made by M on input x. This machine is easily compiled in polynomially steps given $\langle M, x\rangle$. It should be clear that M accepts x if and only if there is an agent that achieves the state $\{accept\}$ in Env_{Mx}. □

The following corollaries are easily obtained via Theorem 1

COROLLARY 1. *AD_{det} is not recursive.*

Proof: The language L_{univ} is not recursive, (see, e.g., [4, p47]), since Theorem 1 shows L_{univ} to be reducible to AD_{det} we deduce that AD_{det} is not recursive. □

COROLLARY 2. *MD_{det} is not recursively enumerable.*

Proof: Consider the language

$$co - L_{univ} = \{\langle M, x\rangle \mid \ M \text{ does } \textit{not} \text{ accept } x\}$$

which is not recursively enumerable. Let $\langle Env_{Mx}, B\rangle$ be the instance of MD_{det} in which Env_{Mx} is defined as in Theorem 1 and $B = \{accept\}$. Then MD_{det} returns true for $\langle Env_{Mx}, B\rangle$ if and only if M does not accept x. □

It is interesting to note that some maintenance design problems within unbounded environments can be solved by *finitely specified* agents for some 'non-trivial' cases.

Example: Set $Env = \langle E, e_0, Ac, \tau\rangle$ where $E = \{0, 1, fail\}$, $e_0 = 0$, $Ac = \{\alpha_0, \alpha_1\}$. For any $s \in S^E$, for which $fail \notin s$, $st(s)$ is the binary word formed by the sequence of states in $\{0, 1\}$ visited by s, e.g. $st(e_0) = 0$. Letting $\{0, 1\}^+$ denote the set of *non-empty* binary words, the state transformer, τ, is given by

$$\tau(r{\cdot}\alpha) = \begin{cases} \emptyset & \text{if} \quad last(r) = fail \\ fail & \text{if} \quad \exists \sigma \in \{0,1\}^*, w \in \{0,1\}^+ \cdot \ st(r) \equiv \sigma www \\ 0 & \text{if} \quad \alpha = \alpha_0 \ \textit{and} \ \forall \sigma \in \{0,1\}^*, w \in \{0,1\}^+, \ st(r) \not\equiv \sigma www \\ 1 & \text{if} \quad \alpha = \alpha_1 \ \textit{and} \ \forall \sigma \in \{0,1\}^*, w \in \{0,1\}^+, \ st(r) \not\equiv \sigma www \end{cases}$$

Now consider the maintenance design instance $CubeFree = \langle Env, \{fail\}\rangle$, requiring the specification of an agent that always avoids the state $\{fail\}$. Notice that any such agent must have unbounded runs and that the *fail* state is forced whenever the same *sequence* of states in $\{0, 1\}$ is visited on three *consecutive* occasions.

Let $\beta_k(\sigma)$ denote the kth bit value in the binary word $\sigma = \sigma_1\sigma_2\cdots\sigma_n$ $(1 \leq k \leq n)$. the agent, C defined for $r \in R_{Env}$ by

$$C(r) = \begin{cases} \alpha_0 & \text{if} \quad |st(r)| \textit{ is odd and } \beta_{(|st(r)|+1)/2}(st(r)) = 1 \\ \alpha_0 & \text{if} \quad |st(r)| \textit{ is even and } \beta_{|st(r)|/2}(st(r)) = 0 \\ \alpha_1 & \text{otherwise} \end{cases}$$

So the sequence traversed by C from $e_0 = 0$, is $01001101001\cdots$. The sequence defined is the so-called Prouhet-Thue-Morse sequence, [10, 12, 14] and can be shown to be *cube free*, i.e. contains no subword of the form www for all $w \in \{0,1\}^+$. This property of the agent, C, suffices to show that it always avoids the state *fail*. It is obvious that the action to be taken by the agent at each stage is easily computable.

We note one historic application of this sequence was given by Euwe [5], in the context of avoiding drawn positions in Chess arising from the 'German rule', under which a game is drawn if the same sequence of moves occurs three times in succession. $\square$

COROLLARY 3. *Define the language $L_{Bounded}$ as,*

$$L_{Bounded} = \{Env \mid \textit{ Env is a bounded environment}\}$$

a) $L_{Bounded}$ is not recursive.

b) $L_{Bounded}$ is recursively enumerable.

Proof: a) The Halting Problem for Turing Machines is easily reduced to $L_{Bounded}$ using the construction of Env_{Mx} in Theorem 1.

b) Given *Env* use a Turing machine program which generates each set $R^{(k)}_{Env}$ for increasing k. If at some point this set is empty, *Env* is accepted. $\square$

THEOREM 2. *AD_X is recursively enumerable.*

Proof: Given an instance $\langle Env, G\rangle$ of $AD_{non\text{-}det}$ we can use (without loss of generality) a non-deterministic Turing machine program, M, that performs that following actions:

1. $k := 0; A_k := \{e_0\}$
2. $A_k := A_k - \{r \in A_k \mid \text{ } r \text{ contains some } g \in G\}$
3. If $A_k = \emptyset$, then accept $\langle Env, G\rangle$.
4. $k := k + 1; A_k := \emptyset;$

5. for each $r \in A_{k-1}$
   Non-deterministically choose $\alpha \in Ac$
   if $\tau(r \cdot \alpha) = \emptyset$ then reject $\langle Env, G\rangle$
   else $A_k := A_k \cup \cup_{e \in \tau(r \cdot \alpha)} \{r \cdot \alpha \cdot e\}$
6. Go to (2).

□

The results above indicate that the decision problems AD_{det} and MD_{det} are provably intractable in the sense that effective decision methods do not exist.

3.2. THE COMPLEXITY OF *FAD* AND *FMD* IN HISTORY DEPENDENT ENVIRONMENTS

The corollary to Theorem 1 rules out any possibility of computationally effective methods existing for AD_X, and MD_X. In this subsection we examine the variants *FAD* and *FMD*. We recall that in these problems only agents which perform *at most* $|E \times Ac|$ actions are considered. We first show that without loss of generality, it will suffice to concentrate on finite achievement problems.

THEOREM 3. *For $X \in \{det, not\text{-}det\}$,*

$$FAD_X \equiv_{\log} FMD_X$$

(i.e., $FAD_X \leq_{\log} FMD_X$ and $FMD_X \leq_{\log} FAD_X$).

Proof: It is unnecessary to prove the two different cases separately, as will be clear from the reductions used. Let,

$$Q_X^{(\epsilon)}(\langle Env, S\rangle) = \begin{cases} FAD_X(\langle Env, S\rangle) & \text{if} \quad \epsilon = 1 \\ FMD_X(\langle Env, S\rangle) & \text{if} \quad \epsilon = 0 \end{cases}$$

For a run r in R_{Env},

$$\Psi^{(\epsilon)}(r, S) = \begin{cases} \exists\, g \in S \text{ such that } g \text{ occurs in } r & \text{if} \quad \epsilon = 1 \\ \forall\, b \in S\ b \text{ does not occur in } r & \text{if} \quad \epsilon = 0 \end{cases}$$

In order to prove the Theorem it suffices to show,

$$\forall \epsilon \in \{0,1\} \;\; Q_X^{(\epsilon)} \;\leq_{\log}\; Q_X^{(\bar{\epsilon})}$$

Let $\langle Env, S\rangle = \langle\langle E, Ac, e_0, \tau\rangle, S\rangle$ be an instance of $Q_X^{(\epsilon)}$. We construct an instance $\langle Env_{(\bar{\epsilon})}, S^{(\bar{\epsilon})}\rangle$ of $Q_X^{(\bar{\epsilon})}$ for which true is returned if and only $Q_X^{(\epsilon)}(\langle Env, S\rangle)$

is true. The instance of $Q_X^{(\bar{\epsilon})}$ has state set $E \cup \{end^{(0)}, end^{(1)}\}$. Herre $end^{(0)}$ and $end^{(1)}$ are new states.The set of actions and start state are as for the instance of $Q_X^{(\epsilon)}$. The set of states $S^{(\bar{\epsilon})}$ is given by $\{end^{(\bar{\epsilon})}\}$. Setting,

$$R^{(0)}_{Env_{(\bar{\epsilon})}} = R^{(0)}_{Env} = \{e_0\}$$

for $r \in R^{(k)}_{Env_{(\bar{\epsilon})}}$ $(k \geq 0)$ and $\alpha \in Ac$,

$$\tau^{(\bar{\epsilon})}(r \cdot \alpha) = \begin{cases} \{end^{(0)}\} & \text{if} \quad r \in C_{Env} \text{ and } \neg\Psi^{(\epsilon)}(r, S) \\ \{end^{(1)}\} & \text{if} \quad r \in C_{Env} \text{ and } \Psi^{(\epsilon)}(r, S) \\ \emptyset & \text{if} \quad last(r) \in \{end^{(0)}, end^{(1)}\} \\ \tau(r \cdot \alpha) & \quad\text{otherwise} \end{cases}$$

First observe that every $r \in R^{(k)}_{Env_{(\bar{\epsilon})}}$ either occurs in $R^{(k)}_{Env}$ or is such that $last(r) \in \{end^{(0)}, end^{(1)}\}$ and has terminated. Furthermore, $Env_{(\bar{\epsilon})}$ is $(|E \times Ac| + 1)$-bounded (even if *Env* is *unbounded*). If $\epsilon = 0$, then $Q_X^{(\bar{\epsilon})}$ is an achievement task with goal state $\{end^{(1)}\}$, otherwise it is a maintenance task with $B = \{end^{(0)}\}$. If *Ag* succeeds with respect to the instance of $Q_X^{(\epsilon)}$ then the agent, $Ag_{(\bar{\epsilon})}$ that follows exactly the same actions as *Ag*, witnesses a positive instance of $Q_X^{(\bar{\epsilon})}$. On the other hand if $Ag_{(\bar{\epsilon})}$ is an agent showing that the instance of $Q_X^{(\bar{\epsilon})}$ should be accepted then all terminated runs r of this agent must have $last(r) = end^{(1)}$. The agent that behaves exactly as $Ag_{(\bar{\epsilon})}$ will always reach some state in S within $|E \times Ac|$ actions ($\epsilon = 1$), respectively avoid every state in S for at least $|E \times Ac|$ actions ($\epsilon = 0$). We observe that: $Env_{(\bar{\epsilon})}$ is deterministic if and only if *Env* is deterministic and uses only 2 additional states. The entire construction can be built in LOGSPACE. □

We note that the construction used in Theorem 3 also applies between *FAD* and *FMD* in the case when the size of critical runs is an arbitrary function of $|E \times Ac|$, e.g. $f(|E \times Ac|) = 2^{|E \times Ac|}$, etc.

THEOREM 4. *If the state transformer function is history dependent, then:*

a) $FAD_{non\text{-}det}$ and $FMD_{non\text{-}det}$ are PSPACE*-complete.*

b) FAD_{det} and FMD_{det} are NP*-complete.*

Proof: As remarked earlier, we need only consider achievement design problems. Recall that only runs in $R^{\leq|E \times Ac|}_{Env}$ are relevant in deciding $FAD_X(\langle Env, G\rangle)$.
Part (a): It must be shown that: (i) $FAD_{non\text{-}det} \in$ PSPACE, and (ii) a known PSPACE-complete problem is polynomially reducible to it.

For (i), we give the design of a non-deterministic polynomial space Turing machine program T that accepts exactly those instances of the problem that have a successful outcome. The inputs to the algorithm will be the task environment $\langle Env, G\rangle$, together with a run $r \in R^{(k)}_{Env}$ — the algorithm actually decides whether or not there is an successful agent continuing from r, i.e., more formally, if there is an agent, Ag, such that every critical run in $C(Ag, Env)$ having r as a *prefix* contains some state $g \in G$

This is first called with r set to the initial state e_0 (the single run in $R^{(0)}_{Env}$). The algorithm for T on input $r \in R^{(k)}_{Env}$ is as follows:

1. if r ends with an environment state in G, then T accepts;
2. if there are no allowable actions given r, then T rejects;
3. if $r \in R^{(|E\times Ac|)}_{Env}$, then T rejects;
4. non-deterministically choose an action $\alpha \in Ac$, and then for each $e \in \tau(r \cdot \alpha)$ recursively call T with the input $r \cdot \alpha \cdot e \in R^{(k+1)}_{Env}$;
5. if all of these accept, then T accepts, otherwise T rejects.

Note that T may safely reject at Step(3): r has not been accepted at Step(1) and any continuation will exceed the total number of actions that an agent is allowed to perform. The algorithm thus non-deterministically explores the space of all possible agents performing at most $|E \times Ac|$ actions, guessing which actions an agent should perform to bring about G. Notice that the depth of recursion will be at most $|E\times Ac|$. Hence T requires only polynomial space. It follows that $FAD_{non\text{-}det}$ is in NPSPACE (i.e., non-deterministic polynomial space). It only remains to note that PSPACE = NPSPACE [11, p150], and so $FAD_{non\text{-}det}$ is also in PSPACE.

For (ii), we must reduce a known PSPACE-complete problem to $FAD_{non\text{-}det}$. The problem we choose is that of determining whether a given player has a winning strategy in the game of generalised geography [11, pp460–462]. We refer to this problem as GG. An instance of GG is a triple $\Gamma = \langle N, A, n\rangle$, where N is a set of nodes, $A \subseteq N \times N$ is a directed graph over N, and $n \in N$ is a node in N. GG is a two player game, in which players I and II take it in turns, starting with I, to select an arc (v, w) in A, where the first arc v must be the “current node”, which at the start of play is n. A move (v, w) changes the current node to w. Players are not allowed to visit nodes that have already been visited: play ends when one player, (the loser), has no moves available. The goal of GG is to determine whether player I has a winning strategy.

GG has a similar structure to AD and we can exploit this to produce a simple mapping from instances $\Gamma = \langle N, A, n\rangle$ of GG to instances $\langle Env, G\rangle$ of $FAD_{non\text{-}det}$. The agent takes the part of player I, the environment takes the part of player II. Begin by setting $E = Ac = N$ and $e_0 = n$. We add a single element e_G to E, and define G to be a singleton containing e_G. These give $|E \times Ac| = (|N| + 1)|N|$. We now need to define τ, the state transformer function of the environment; the idea is to directly encode the arcs of Γ into τ. For $r \in R_{Env}^{(\leq(|N|+1)|N|)}$ and $v \in Ac = N$,

$$\tau(r \cdot v) = \begin{cases} \emptyset & \text{if } (last(r), v) \notin A \\ \{e_G\} & \text{if } \{w \mid (v, w) \in A \text{ and } w \notin r\} = \emptyset \\ \{w \mid (v, w) \in A \text{ and } w \notin r\} & \text{otherwise.} \end{cases}$$

This construction requires a little explanation. The first case deals with the case where the agent has made an illegal move, in which case the environment disallows any further moves: the game ends without the goal being achieved. The second case is where player I (represented by the agent) wins, because there are no moves left for player II. In this case the environment returns G, indicating success. The third is the general case, where the environment returns states corresponding to all possible moves. Using this construction, there will exist an agent that can succeed in the environment we construct just in case player I has a winning strategy for the corresponding GG game. Since nodes (i.e states in E) cannot be revisited any successful agent, Ag, satisfies $R(Ag, Env) \subseteq R_{Env}^{(\leq N+1)}$. Since the construction clearly takes polynomial time, it follows that $FAD_{non\text{-}det}$ is complete for PSPACE, and we are done.

Part (b): For part (b), we first show that $FAD_{det} \in$ NP. The non-deterministic polynomial time algorithm for deciding the problem is as follows. Given an instance $\langle Env, G \rangle$ of FAD_{det}, non-deterministically construct a run $r \in R_{Env}^{(\leq|E \times Ac|)}$: this can be done in non-deterministic polynomial time using T_τ. The instance of FAD_{det} is accepted if r contains some $g \in G$.

To prove that the problem is NP-hard we give a reduction from the directed Hamiltonian cycle (DHC) problem to FAD_{det} [11, p209]. An instance of DHC is given by a directed graph $H = \langle V, F \subseteq V \times V \rangle$; the aim is to determine whether or not H contains a directed Hamiltonian cycle.

The idea of the reduction is to encode the graph H directly in the state transformer function τ of the environment: actions correspond to edges of the graph, and success occurs when a Hamiltonian cycle has been found.

Formally, given an instance $H = \langle V, F \subseteq V \times V \rangle$ of DHC, we generate an instance of FAD_{det} as follows. First, create the set of environment states and initial state as follows:

$$E = V \cup \{succeed\}; \qquad e_0 = v_0$$

We create an action $\alpha_{i,j}$ corresponding to every arc in H, i.e., $Ac = \{\alpha_{i,j} \mid \langle v_i, v_j \rangle \in F\}$. The set G is defined to be a singleton: $G = \{succeed\}$. With these, $|E \times Ac| = (|V| + 1)|F|$. Finally, we define τ in two parts. The first case deals with the first action of the agent:

$$\tau(e_0 \cdot \alpha_{0,j}) = \begin{cases} v_j & \text{if } \langle v_0, v_j \rangle \in F \\ \emptyset & \text{otherwise.} \end{cases}$$

The second case deals with subsequent actions:

$$\tau(r \cdot v_i \cdot \alpha_{i,j}) = \begin{cases} \emptyset & \text{if } v_j \text{ occurs in } r \cdot v_i \text{ and } v_j \neq v_0 \\ \mathit{succeed} & \text{if } v_j = v_0 \text{ and every } v \in V \text{ occurs in } r \cdot v_i \\ v_j & \text{if } \langle v_i, v_j \rangle \in F \end{cases}$$

An agent can only succeed in this environment if it visits every vertex of the original graph. An agent will fail if it revisits any vertex. Thus, for any successful agent, Ag, $R(Ag, Env) \subseteq R_{Env}^{(\leq |V|+1)}$. Since the construction is clearly polynomial time, we are done. (As an aside, notice that the environment created in the reduction is history dependent.)

□

We conclude this section noting one generalisation of Theorem 4(b).

Let $\beta(n,k)$ be the function defined over $I\!N \times I\!N$ by $\beta(n,0) = n$ and $\beta(n,k) = 2^{\beta(n,k-1)}$ (for $k \geq 1$). The complexity class, NEXP^{k}TIME is

$$\text{NEXP}^k\text{TIME} = \bigcup_{p=0}^{\infty} \text{NTIME}(\beta(n^p, k))$$

(hence, NP $\equiv$ NEXP^{0}TIME, etc)

Let $FAD^{(k)}$ be the extension of FAD, in which runs in $R_{Env}^{(\leq \beta(|E \times Ac|, k))}$ are of interest (so that $FAD \equiv FAD^{(0)}$).

THEOREM 5. $\forall k \geq 0$ $FAD_{det}^{(k)}$ *is* NEXP^{k}TIME-*complete.*

Proof: Omitted. (given in Appendix) □

3.3. The Complexity of *FAD* and *FMD* in History Independent Environments

The definition of state transformer function (whether non-deterministic or deterministic) is as a mapping from $r \in R^{Ac}$ to (sets of) states. In this way the outcome of an action may depend not only on the current state but also on the previous *history* of the environment that led to the current state. We now consider the agent design decision problems examined above, but in the context of *history independent* state transformer functions. In the non-deterministic case, a history independent environment has a state transformer function with the signature

$$\tau : E \times Ac \to \wp(E)$$

and, in deterministic environments

$$\tau : E \times Ac \to E \cup \{\emptyset\}.$$

In such environments, τ is represented as a directed graph $H_\tau(V,A)$ in which each state $e \in E$ labels a single vertex in V and for which there is an edge $\langle u,w\rangle \in A$ labelled $\alpha \in Ac$ if $w \in \tau(u,\alpha)$ (non-deterministic case) or $w = \tau(u,\alpha)$ (deterministic). We note that, by retaining the requirment for environments to be bounded, so that $R_{Env}^{(|V|+1)} = \emptyset$, without loss of generality, we can regard $H_\tau(V,A)$ as *acyclic*.

THEOREM 6. *If the state transformer function is history independent, then:*

a) FAD_{det} and FMD_{det} are NL*-complete.*

b) $FAD_{non\text{-}det}$ and $FMD_{non\text{-}det}$ are P*-complete.*

Proof: For part (a), deciding a history-independent instance $\langle Env, \{succeed\}\rangle$ of AD_{det} is equivalent to deciding if there is a directed path in H_τ from e_0 to *succeed*. This is identical to the well-known NL-complete problem Graph Reachability [11, p398].

For part (b) that $FAD_{non\text{-}det} \in$ P follows from the algorithm $Label(v)$, below, that given $H(V,A)$ and $G \subseteq V$ labels each vertex, v, of H with "true" or "false" according to whether a successful agent can be found starting from v.

Initially all vertices, v, have $label(v)$ unassigned.

1) If v has been labelled, then return $label(v)$.

2) If $v \in G$, $label(v) := true$; return true.

3) $Avail(v) := \{\alpha \mid \exists w \text{ with } (v \to_\alpha w) \in A\}$

4) If $Avail(v) = \emptyset$, $label(v) := false$; return false;

5) Choose $\alpha \in Avail(v)$; $Avail(v) := Avail(v) \setminus \{\alpha\}$;

6) $label(v) = \bigwedge_{\{w \mid v \to_\alpha w \in A\}} label(w)$;

7) If $label(v) = true$ then return true;

8) go to (4)

That this algorithm is polynomial-time follows from the fact that no edge of H is inspected more than once. A similar method, can be applied for $FMD_{non\text{-}det}$.

We now show that the MONOTONE CIRCUIT VALUE PROBLEM ($MCVP$) is reducible in LOGSPACE to $FAD_{non\text{-}det}$. An instance of $MCVP$ consists of a sequence of $m+n$ triples $S = \langle t_1, t_2, \cdots, t_n, t_{n+1}, \cdots, t_{m+n}\rangle$. The triples t_i for $1 \leq i \leq n$ are called *inputs* and have the form $(x_i, 0, 0)$ with $x_i \in \{true, false\}$. The remaining triples are called *gates* and take the form (op_i, j, k), with $op_i \in \{\wedge, \vee\}$, $j < i$ and $k < i$. A Boolean value, $\nu(i)$ is defined for each t_i as follows:

$$\nu(i) = \begin{cases} x_i & \text{if} \quad t_i \text{ is an input of } S \\ \nu(j) \vee \nu(k) & \text{if} \quad t_i = (\vee, j, k) \\ \nu(j) \wedge \nu(k) & \text{if} \quad t_i = (\wedge, j, k) \end{cases}$$

An instance, S, is accepted if $\nu(m+n) = true$. $MCVP$ was shown to be P-complete in [6].

Given an instance $S = \langle t_1, \cdots, t_n, t_{n+1}, \cdots, t_{n+m}\rangle$ of $MCVP$ we construct an instance $\langle Env_S, G_S\rangle$ of $FAD_{non\text{-}det}$ for which $Env_S = \langle V_S, Ac_S, v_0, H_S(V_S, A_S)\rangle$ is history-independent and non-deterministic, $H_S(V_S, A_S)$ being a directed acyclic graph whose edges, A_S are labelled with actions from Ac_S. The construction will use working space that is logarithmic in the number of bits needed to encode S.

We set, $V_S = \{q_1, q_2, \cdots, q_n, q_{n+1}, \cdots, q_{n+m}\} \cup \{accept\}$, $Ac_S = \{\alpha, \beta\}$, $v_0 = q_{n+m}$, and $G = \{accept\}$. The edges A_S and associated actions are given by

$$\begin{array}{lll} (q_i \rightarrow_\alpha accept) & \text{if} & t_i = (true, 0, 0) \\ (q_i \rightarrow_\alpha q_j),\ (q_i \rightarrow_\alpha q_k) & \text{if} & t_i = (\wedge, j, k) \\ (q_i \rightarrow_\alpha q_j),\ (q_i \rightarrow_\beta q_k) & \text{if} & t_i = (\vee, j, k) \end{array}$$

We claim that,

$$\nu(i) = true \Leftrightarrow \text{There is an agent that succeeds in reaching } accept \text{ starting in } q_i$$

We prove this by induction on i. For the inductive base, $1 \leq i \leq n$, $\nu(i) = true$ if the input t_i has the form $(true, 0, 0)$. In this case there is an edge labelled α from the corresponding state q_i to the state *accept*. Similarly, if there *is* an edge $(q_i \rightarrow_\alpha accept)$ in A_S this could only be present though t_i being $(true, 0, 0)$, i.e., $\nu(i) = true$.

Assuming the claim holds whenever $1 \leq i \leq n+p$, we show it holds for $i > n+p$. Consider the triple t_{n+p+1}. This must be a gate in S, $t_{n+p+1} = (op_{n+p+1}, j, k)$. Since $j, k < n+p+1$, from the Inductive Hypothesis,

$$\nu(j) = true \Leftrightarrow \text{There is an agent that succeeds in reaching } accept \text{ starting in } q_j$$
$$\nu(k) = true \Leftrightarrow \text{There is an agent that succeeds in reaching } accept \text{ starting in } q_k$$

If $op_{n+p+1} = \wedge$ then $\nu(n+p+1) = true$ only if both of $\nu(j)$ and $\nu(k)$ equal *true*. Choosing the action α in state q_{n+p+1} would thereby yield an agent achieving *accept*. In the same way, if there is an agent starting from q_{n+p+1} that succeeds, since α is the only action available, it must be the case that successful agents can be defined from both of q_j and q_k. Applying the Inductive Hypothesis, we deduce that $\nu(n+p+1) = \nu(j) \wedge \nu(k) = true$.

If $op_{n+p+1} = \vee$ then $\nu(n+p+1) = true$ only if at least one of $\nu(j)$, $\nu(k)$ equals *true*. If $\nu(j) = true$ the action α can be chosen to succeed from q_{n+p+1}, otherwise the action β can be selected. To complete the inductive step we observe that that if q_{n+p+1} leads to a successful agent then this agent must either pass through q_j (α chosen) or q_k (β chosen), thus $\nu(n+p+1) = \nu(j) \vee \nu(k) = true$.

Recalling that $v_0 = q_{n+m}$, it follows that

$$\nu(n+m) = true \Leftrightarrow \text{There is an agent that reaches } accept \text{ from } v_0$$

as required.

By a similar construction we may show that $MCVP \leq_{\log} FMD_{non\text{-}det}$: the state *accept* is replaced by a state *reject* with edges labelled α directed into this from any q_i for which $t_i = (false, 0, 0)$. S is accepted by $MCVP$ if and only if an agent can be defined from q_{n+m} that always avoids the state $\{reject\}$. □

4. Conclusions and Related Work

In this article, we have shown that, depending on the properties of the environment, the agent design problem ranges from undecidable (not recursively enumerable, in the case of the maintenance design problem in unbounded environments) to tractable (NL-complete, in the case of achievement and maintenance design problems in deterministic, history independent environments).

In the AI literature, the most closely related work to our own is on the complexity of the planning problem [1]. Bylander was probably the first to undertake a systematic study of the complexity of the planning problem; he showed that the (propositional) STRIPS planning problem is PSPACE-complete [3]. Building on his work, many other variants of the planning problem have been studied — recent examples include [9, 2]. The main differences between our work and the work on AI planning is as follows:

– The notion of an agent in our work (as a function that maps runs to selected actions) is more general than the notion of a plan as it commonly appears in the planning literature. Our agents are more akin to the notion of a *strategy* in game theory. The obvious advantage of our approach is that our results are not bound to a particular plan representation. The obvious disadvantage is that having a positive answer to one of our agent design problems (e.g., knowing that there exists an agent to carry out a task in some environment) does not imply that an agent to carry out the task will be *implementable*, in the sense of [13].

– Most complexity results in the planning literature are bound to particular *representations* of goals and actions. The STRIPS notation in Bylander's work is one example [3]; Baral *et al.* use the action description language $\mathcal{A}$ [2]; in the work of Littman *et al*, the representation chosen is ST [9]. In some cases, it is not clear whether results reflect the complexity of the decision problem, or whether they are at least in part an artifact of the representation. We have adopted the most general representations possible, for example representing achievement and maintenance tasks through sets of states, rather than through a particular (e.g., logical) formalism, and consider several possible representations of the environment.

– Most complexity results in planning assume simple (deterministic, history independent) environments.

– The focus in the planning literature has been almost exclusively on achievement tasks. We have also considered maintenance tasks, and elsewhere, have considered richer task specifications [16].

In the computational complexity literature, the most relevant problems are those of determining whether or not a given player has a winning strategy in a particular two-player game. PSPACE-completeness appears to be the characteristic complexity result for such problems [11, pp459–480].

There are many issues that demand attention in future work. One is the relationship of our work to that of solving Markov decision problems [8]. Another key problem is that of determining the extent to which our positive (polynomial time) results can be exploited in practice. Yet another is on extending our task specification framework to allow richer and more complex tasks.

References

1. J. F. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann Publishers: San Mateo, CA, 1990.
2. C. Baral, V. Kreinovich, and R. Trejo. Computational complexity of planning and approximate planning in presence of incompleteness. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, 1999.
3. T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.
4. P.E. Dunne. *Computability Theory - Concepts and Applications*. Ellis-Horwood, 1991.

5. M. Euwe. Mengentheoretische Betrachten über das Schachspiel *Proc. Konin. Akad. Wetenschappen, Amsterdam*, 32, 633–642, 1929
6. L.M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT Newsletter*, 9(2):25–29, 1977.
7. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
8. L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.
9. M. L. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of AI Research*, 9:1–36, 1998.
10. M. Morse. Abstract 360: a solution of the problem of infinite play in chess. *Bull. Amer. Math. Soc.*, 44, page 632, 1938
11. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley: Reading, MA, 1994.
12. E. Prouhet. Mémoire sur quelques relations entre les puissances des nombres. *R. Acad. Sci. Paris Sér. I*, 33, page 225, 1851.
13. S. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of AI Research*, 2:575–609, 1995.
14. A. Thue. Über unendliche Zeichenreihen. *Norske vid. Selsk. Skr. Mat. Nat. Kl.*, 7, pages 1–22, 1906. (Republished in: "Selected Mathematical Papers of Axel Thue," (ed: T. Nagell, Universitetsforlaget, Oslo, pages 139–158, 1977
15. M. Wooldridge. The computational complexity of agent design problems. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, pages 341–348, Boston, MA, 2000.
16. M. Wooldridge and P. E. Dunne. Optimistic and disjunctive agent design problems. In Y. Lespérance and C. Castelfranchi, editors, *Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, 2000.
17. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Appendix: Proof of Theorem 14

The decision problem denoted $FAD^{(k)}$ takes as input a pair $\langle Env, G\rangle$ returning true if and only if there is an agent, Ag, with which,

$$\forall r \in T^{(\leq\beta(|E\times Ac|,k-1))}(Ag,Env)\cup R^{(\beta((|E\times Ac|,k))}(Ag,Env)\exists g \in G \text{ such that } g \text{ is in } r$$

Recall that Theorem 14 stated:

$$\forall k \geq 0 \;\; FAD^{(k)}_{det} \text{ is } \mathrm{NEXP}^k\mathrm{TIME}\text{-complete}$$

Proof: That $FAD^{(k)}_{det} \in \mathrm{NEXP}^k\mathrm{TIME}$ follows by using a Turing machine program that non-deterministically guesses a run r in $R^{(\leq\beta(|E\times Ac|,k)}_{Env}$ and then checks whether r contains some state of G.

To complete the proof, we use a *generic reduction* for languages in $L \in \mathrm{NEXP}^k\mathrm{TIME}$ to $FAD^{(k)}_{det}$.

Let L be any language in $\mathrm{NEXP}^k\mathrm{TIME}$. Without loss of generality, it may be assumed that instance of L are encoded using the binary alphabet $\{0,1\}$. The choice of L means that there is a non-deterministic Turing machine program (*NTM*), M, with the following properties:

M1) For any $x \in L$, there is an accepting computation of M on x.

M2) There is a function $q : \mathbb{N} \to \mathbb{N}$, such that $q(n) \leq \beta(n^p, k)$ and M takes at most $q(|x|)$ moves to accept any $x \in L$.

It follows that deciding $x \in L$ reduces to the decision problem

$$NCOMP^{(k)}_M = \{x \mid M \text{ has an accepting computation on } x \text{ using} \leq q(|x|) \text{ moves}\}$$

Without loss of generality we may assume that,

a) M on input x uses a single 2-way infinite tape.

b) The non-deterministic state transition function of M prescribes *exactly* two possible choices of move (except for halting states).

Let x be an instance of $NCOMP^{(k)}_M$ with

$$\begin{array}{l} M = (Q,\Sigma,\Gamma,q_0,\delta,\{q_A,q_R\}) \\ \Sigma = \{0,1\},\ \Gamma = \{0,1,B\} \\ \delta\ :\ Q\times\Gamma \to \wp(Q\times\Sigma\times\{L,R\}) \\ x = x_1x_2\cdots x_n\ \in\ \{0,1\}^n \end{array}$$

We construct an instance $\langle Env_{Mx}, G\rangle$ of $FAD^{(k)}_{det}$ for which

$$\begin{array}{rcl} Env_{Mx} & = & \langle E_{Mx}, Ac_{Mx}, e_0, \tau_{Mx}\rangle \\ x \in NCOMP^{(k)}_M & \iff & FAD^{(k)}_{det}(Env_{Mx}, G) \text{ is true} \end{array}$$

The state set and actions of Env_{Mx} are,

$$\begin{aligned} E_{Mx} &= Q \times \Gamma \ \cup \ \{\langle write, x_i\rangle \mid \ 1 \leq i \leq |x|\} \cup \{e_0\} \\ Ac_{Mx} &= \Sigma \times \{L, R\} \ \cup \ \{copy\} \end{aligned}$$

Finally E_{Mx} is padded with $|x|^p$ inaccessible states in order to ensure that,

$$\beta(|E_{Mx} \times Ac_{Mx}|, k) \geq \beta(|x|^p, k) = q(|x|)$$

(this will only increase the size of the instance $\langle Env_{Mx}, G\rangle$ polynomially).

We define τ_{Mx} so that the possible runs of the environment simulate the moves of M when started with input x. To start the input x is "copied" so that it can be recovered from any run. The action *copy* of Ac_{Mx} is used for this purpose. For $r \in R_{Env_{Mx}}$,

$$\tau_{Mx}(r \cdot copy) = \begin{cases} \langle write, x_{i+1}\rangle & \text{if} \quad r \in R^{(i)}_{Env_{Mx}} \text{ and } i < |x| \\ \langle q_0, B\rangle & \text{if} \quad r \in R^{(0)}_{Env_{Mx}} \text{ and } |x| = 0 \\ \langle q_0, x_1\rangle & \text{if} \quad r \in R^{(|x|)}_{Env_{Mx}} \\ \emptyset & \text{if} \quad r \notin R^{(\leq|x|)}_{Env_{Mx}} \end{cases}$$

The *copy* action is the only allowable action for runs in $R^{(\leq|x|)}_{Env_{Mx}}$, i.e.

$$\forall \alpha \in Ac_{Mx} \setminus \{copy\} \ \forall r \in R^{(\leq|x|)}_{Env_{Mx}} \ \tau_{Mx}(r \cdot \alpha) = \emptyset$$

To complete the description of τ_{Mx}, we define its behaviour on runs in $R^{(i)}_{Env_{Mx}}$ when $i > |x|$.

The main idea is that any run r in $R^{(|x|+i)}_{Env_{Mx}}$ will encode *one* possible computation of M on x after M has made exactly $i-1$ moves. Thus if M has an accepting computation on x, an encoding of this will appear in $R^{(\leq\beta(|E_{Mx}\times Ac_{Mx}|,k)}_{Env_{Mx}}$.

In order to do this, the idea of the *configuration of* M, associated with a run r is needed. This is denoted $\chi(r)$ and defined for any run in $R^{(|x|+i)}_{Env_{Mx}}$, when $i > 0$. Such a configuration, describes the (non-blank) portion of M's tape, together with its current state and head position.

For $r \in R^{(|x|+1)}_{Env_{Mx}}$,

$$\chi(r) = \begin{cases} q_0 B & \text{if} \quad |x| = 0 \\ q_0 x_1 x_2 \cdots x_n & \text{if} \quad |x| > 0 \end{cases}$$

Now, suppose $r \in R^{(|x|+i)}_{Env_{Mx}}$ with $i > 0$ and that

$$\chi(r) \in \left\{ \begin{array}{l} q_i B c_1 c_2 \cdots c_m \\ c_1 c_2 \cdots c_m q_i B \\ c_1 c_2 \cdots c_{t-1} q_i \gamma c_{t+1} \cdots c_m \end{array} \right\}$$

where $c_j \in \{0,1\}$ $(1 \le j \le m)$. The case $m = 0$ corresponds to the input word x being empty. In each case M would be in state q_i. For the first two possibilities, the head is scanning the blank symbol immediately to the left (respectively, right) of the contiguous block of non-blank symbols $c_1 c_2 \cdots c_m$. In the third case, some non-blank symbol γ is being scanned.

If $q_i \in \{q_A, q_R\}$, then $\tau_{Mx}(r \cdot \alpha) = \emptyset$ for all $\alpha \in Ac_{Mx}$.

Otherwise, the state transition function, δ of M prescribes *exactly two* possible moves, i.e

$$\begin{array}{lll}\delta(q_i, B) = \{\langle q_j^{(1)}, \sigma^{(1)}, D^{(1)}\rangle, \langle q_j^{(2)}, \sigma^{(2)}, D^{(2)}\rangle\} & \text{if} & \chi(r) \in \{q_iBc_1\cdots c_m, c_1\cdots c_m q_i B\} \\ \delta(q_i, \gamma) = \{\langle q_j^{(1)}, \sigma^{(1)}, D^{(1)}\rangle, \langle q_j^{(2)}, \sigma^{(2)}, D^{(2)}\rangle\} & \text{if} & \chi(r) = c_1\cdots c_{t-1} q_i \gamma c_{t+1} \cdots c_m\end{array}$$

In these cases the only allowable actions in Ac_{Mx} are

$$\langle \sigma, D\rangle \in \{\langle \sigma^{(1)}, D^{(1)}\rangle, \langle \sigma^{(2)}, D^{(2)}\rangle\}$$

and

$$\tau_{Mx}(r\cdot\langle\sigma,D\rangle) = \begin{cases} \langle q_j, B\rangle & \text{if} \quad D = L \text{ and } \chi(r) = q_iBc_1\cdots c_m \\ \langle q_j, c_1\rangle & \text{if} \quad D = R \text{ and } \chi(r) = q_iBc_1\cdots c_m \\ \langle q_j, c_m\rangle & \text{if} \quad D = L \text{ and } \chi(r) = c_1\cdots c_m q_i B \\ \langle q_j, B\rangle & \text{if} \quad D = R \text{ and } \chi(r) = c_1\cdots c_m q_i B \\ \langle q_j, c_{t-1}\rangle & \text{if} \quad D = L \text{ and } \chi(r) = c_1\cdots c_{t-1} q_i\gamma c_{t+1}\cdots c_m \ (t > 1) \\ \langle q_j, B\rangle & \text{if} \quad D = L \text{ and } \chi(r) = c_1\cdots c_{t-1} q_i\gamma c_{t+1}\cdots c_m \ (t = 1) \\ \langle q_j, c_{t+1}\rangle & \text{if} \quad D = R \text{ and } \chi(r) = c_1\cdots c_{t-1} q_i\gamma c_{t+1}\cdots c_m \ (t < m) \\ \langle q_j, B\rangle & \text{if} \quad D = R \text{ and } \chi(r) = c_1\cdots c_{t-1} q_i\gamma c_{t+1}\cdots c_m \ (t = m) \end{cases}$$

with the new configurations being,

$$\chi(r\cdot\langle\sigma,D\rangle\cdot e) = \begin{cases} q_jB\sigma c_1\cdots c_m & \text{if} \quad D = L,\ \chi(r) = q_iBc_1\cdots c_m, \\ & \quad\ \text{and } e = \langle q_j, B\rangle \\ \sigma q_j c_1\cdots c_m & \text{if} \quad D = R,\ \chi(r) = q_iBc_1\cdots c_m, \\ & \quad\ \text{and } e = \langle q_j, c_1\rangle \\ c_1\cdots q_j c_m \sigma & \text{if} \quad D = L,\ \chi(r) = c_1\cdots c_m q_i B, \\ & \quad\ \text{and } e = \langle q_j, c_m\rangle \\ c_1\cdots c_m \sigma q_j B & \text{if} \quad D = R,\ \chi(r) = c_1\cdots c_m q_i B, \\ & \quad\ \text{and } e = \langle q_j, B\rangle \\ c_1\cdots q_j c_{t-1}\sigma c_{t+1}\cdots c_m & \text{if} \quad D = L,\ \chi(r) = c_1\cdots c_{t-1} q_i\gamma c_{t+1}\cdots c_m, \\ & \quad\ \text{and } e = \langle q_j, c_{t-1}\rangle \ (t > 1) \\ q_jB\sigma c_2\cdots c_m & \text{if} \quad D = L,\ \chi(r) = q_i\gamma c_2\cdots c_m, \\ & \quad\ \text{and } e = \langle q_j, B\rangle \ (t = 1) \\ c_1\cdots c_{t-1}\sigma q_j c_{t+1}\cdots c_m & \text{if} \quad D = R,\ \chi(r) = c_1\cdots c_{t-1} q_i\gamma c_{t+1}\cdots c_m, \\ & \quad\ \text{and } e = \langle q_j, c_{t+1}\rangle \ (t < m) \\ c_1\cdots c_{m-1}\sigma q_j B & \text{if} \quad D = R,\ \chi(r) = c_1\cdots c_{m-1} q_i\gamma, \\ & \quad\ \text{and } e = \langle q_j, B\rangle \ (t = m) \end{cases}$$

Given $r \in R_{Env_{Mx}}$ the configuration $\chi(r)$ can be constructed in $O(|r|)$ steps. To complete the construction we set $G = \{q_A\} \times \Gamma$.

We claim that $x \in NCOMP_M^{(k)}$ if and only if "true" is returned for the instance $\langle Env_{Mx}, G\rangle$ of $FAD_{det}^{(k)}$ defined above.

Suppose $x \in NCOMP_M^{(k)}$. There must be a sequence $\langle \psi_0, \psi_1, \cdots, \psi_t\rangle$ of configurations (of M on input x) for which ψ_{i+1} results from ψ_i after a single move of M, the state indicated in ψ_t is q_A and $t \leq \beta(|x|^p, k)$. The construction of Env_{Mx} shows that we can chose a sequence of actions for an agent Ag, with which the configuration of M, $\chi(r)$ associated with the run $r \in R^{(|x|+1)}(Ag, Env_{Mx})$ is ψ_0. Furthermore, the run $r_i \in R^{(|x|+1+i)}(Ag, Env_{Mx})$ satisfies $\psi_i = \chi(r_i)$. Thus, the single terminated run r of Ag has $last(r)$ equal to some state $\langle q_A, \gamma\rangle$ of E_{Mx}. We deduce that the sequence of actions used to progress from the run $\{e_0\} \in R_{Env_{Mx}}^{(0)}$ through to the (terminated) run $r_{|x|+1+t} \in R^{(|x|+1+t)}(Ag, Env_{Mx})$ defines an agent accomplishing the achievement task specified by G. It remains only to note that $|x| + 1 + t \leq \beta(|E_{Mx} \times Ac_{Mx}|, k)$.

In a similar manner, if $\langle Env_{Mx}, G\rangle$ is a positive instance of $AD_{det}^{(k)}$ then the sequence of actions performed by a successful agent, Ag, from the run in $R^{(|x|+1)}(Ag, Env_{Mx})$ corresponds to a sequence of moves made by M that reach the accept state q_A.

To complete the proof it suffices to observe that, a Turing machine program for τ_{Mx} can be easily compiled from the Turing machine, M, and its input x. □