Optimistic and Disjunctive Agent Design Problems

Michael Wooldridge and Paul E. Dunne

Department of Computer Science
University of Liverpool
Liverpool L69 7ZF
United Kingdom
{M.J.Wooldridge, P.E.Dunne}@csc.liv.ac.uk

Abstract. The *agent design* problem is as follows: Given an environment, together with a specification of a task, is it possible to construct an agent that will guarantee to successfully accomplish the task in the environment? In previous research, it was shown that for two important classes of tasks (where an agent was required to either achieve some state of affairs or maintain some state of affairs), the agent design problem was PSPACE-complete. In this paper, we consider several important generalisations of such tasks. In an *optimistic* agent design problem, we simply ask whether an agent has at least *some* chance of bringing about a goal state. In a *combined* design problem, an agent is required to achieve some state of affairs while ensuring that some invariant condition is maintained. Finally, in a *disjunctive* design problem, we are presented with a number of goals and corresponding invariants — the aim is to design an agent that on any given run, will achieve one of the goals while maintaining the corresponding invariant. We prove that while the optimistic achievement and maintenance design problems are NP-complete, the PSPACE-completeness results obtained for achievement and maintenance tasks generalise to combined and disjunctive agent design.

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 paper, however, we focus on the underlying decision problems associated with the deployment of such agents. Specifically, we study the *agent design* problem [16].

The agent design problem may be stated as follows: Given 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 previous research, it was shown that for two important classes of tasks (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), the agent design problem is PSPACE-complete in the most general case [16].

In this paper, we consider several important variations of such tasks. First, in an *optimistic* agent design problem, we simply ask whether there exists an agent that has at least *some chance* of achieving the goal or maintaining the condition respectively. In a *combined* agent design problem, the task involves achieving some state of affairs *while at the same time* ensuring that some invariant condition is maintained. In a *disjunctive* design problem, we are presented with a number of goals and corresponding invariants — the aim is to design an agent that on any given run, will achieve one of the goals while maintaining the corresponding invariant. We prove that for optimistic achievement and maintenance tasks, the agent design problem is NP-complete, while the PSPACE-completeness results obtained for achievement and maintenance tasks generalise to combined and disjunctive agent design problems.

We begin in the following section by setting up an abstract model of agents and environments, which we use to formally define the decision problems under study. We then informally motivate and introduce the various agent design problems we study, and prove our main results. We discuss related work in section 6, and present some conclusions in section 7.

Notation: We use standard set theoretic and logical notation wherever possible, augmented as follows. If S is a set, then the set of finite sequences over S is denoted by S^*. If $\sigma \in S^*$ and $s \in S$, then the sequence obtained by appending s to σ is denoted $\sigma \cdot s$. We write $s \in \sigma$ to indicate that element s is present in sequence σ, and write $last(\sigma)$ to denote the final element of σ. Throughout the paper, we assume some familiarity with complexity theory [12].

2 Agents and Environments

In this section, we present an abstract formal model of agents and the environments they occupy; we then use this model to frame the decision problems we study. 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 $E = \{e, e', \ldots\}$ of instantaneous states. Agents are assumed to have a repertoire of possible actions available to them, which transform the state of the environment. Let $Ac = \{\alpha, \alpha', \ldots\}$ be the (finite) set of actions.

The basic model of agents interacting with their environments is as follows. The environment starts in some state, and the agent begins by choosing an action to perform on that state. As a result of this action, the environment can respond with a number of possible states. However, only one state will *actually* result — though of course, the agent does not know in advance which it will be. On the basis of this second state, the agent again chooses an action to perform. The environment responds with one of a set of possible states, the agent then chooses another action, and so on.

A *run*, r, of an agent in an environment is thus a sequence of interleaved environment states and actions:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{u-1}} e_u \cdots$$

Let $\mathcal{R}$ be the set of all such possible runs. We use $r, r', \ldots$ to stand for members of $\mathcal{R}$.

In order to represent the effect that an agent's actions have on an environment, we introduce a *state transformer* function (cf. [5, p154]):

$$\tau : \mathcal{R} \rightarrow 2^E$$

Thus a state transformer function maps a run (assumed to end with the action of an agent) to a set of possible environment states. There are two important points to note about this definition.

First, environments are allowed to be *history dependent* (or non-Markovian). In other words, the next state of an environment is not solely determined by the action performed by the agent and the current state of the environment. The previous actions performed by the agent, and the previous environment states also play a part in determining the current state. Many environments have this property. For example, consider the well-known travelling salesman problem [12, p13]: history dependence arises because the salesman is not allowed to visit the same city twice. Note that it is often possible to transform a history dependent 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 exponential increase in the number of environment states. Intuitively, given a history dependent environment with state set E, which has an associated set of runs $\mathcal{R}$, we would need to create $|\mathcal{R} \times E|$ environment states. Since $|\mathcal{R}|$ is easily seen to be exponential in the size of $Ac \times E$ (even if we assume a polynomial bound on the length of runs), this implies that the transformation is not likely to be possible in polynomial time (or space). Hence although such a transformation is possible in principle, it is unlikely to be possible in practice.

Second, note that this definition allows for non-determinism in the environment. There is thus *uncertainty* about the result of performing an action in some state.

If $\tau(r) = \emptyset$, (where r is assumed to end with an action), then there are no possible successor states to r. In this case, we say that there are *no allowable actions*, and that the run is *complete*. One important assumption we make is that every run is guaranteed to complete with length polynomial in the size of $Ac \times E$. This assumption may at first sight appear restrictive, and so some justification is necessary. Our main point is that exponential (or worse) runs *are of no practical interest whatsoever*: the *only* tasks of interest to us are those that require a polynomial (or better) number of actions to achieve. To see this, suppose we allowed runs that were exponential in the size of $Ac \times E$; say $O(2^{|Ac \times E|})$. Now consider a trivial environment, with just 10 states and 10 actions. Then in principle, such an environment would allow tasks that require $2^{100} = 1.2 \times 10^{30}$ actions to accomplish. Even if our agents could perform a 10^9 actions per second[1] then such a task would require more time to carry out than there has been since the universe began. The exponential length of runs will rapidly eliminate any advantage we gain by multiplying the speed of our agent by a constant factor. The polynomial restriction on run length is, therefore, entirely reasonable if we are concerned with tasks of practical interest.

Before proceeding, we need to make clear a couple of assumptions about the way that transformer functions are *represented*. To understand what is meant by this, con-

[1] A high performance desktop computer can carry out about this many operations per second.

sider that the input to the decision problems we study will include some sort of representation of the behaviour of the environment, and more specifically, the environment's state transformer function τ. Now, one possible description of τ is as a table that maps run/action pairs to the corresponding possible resulting environment states:

$$\begin{array}{ll} r_1, \alpha_1 & \rightarrow \{e_1, e_2, \ldots\} \\ \ldots & \rightarrow \ldots \\ r_n, \alpha_n & \rightarrow \{\ldots\} \end{array}$$

Such a "verbose" encoding of τ will clearly be exponentially large (in the size of $E \times Ac$), but since the length of runs will be bounded by a polynomial in the size of $E \times Ac$, it will be finite. Once given such an encoding, finding an agent that can be guaranteed to achieve a set of goal states will, however, be comparatively easy. Unfortunately, of course, no such description of the environment will usually be available. In this paper, therefore, we will restrict our attention to environments whose state transformer function is described as a two-tape Turing machine, with the input (a run and an action) written on one tape; the output (the set of possible resultant states) is written on the other tape. It is assumed that to compute the resultant states, the Turing machine requires a number of steps that is at most polynomial in the length of the input. We refer to such environment representations as *concise*. In the remainder of this paper, we will assume that all state transformer functions are concisely represented.

Formally, we say an environment Env is a triple $Env = \langle E, \tau, e_0 \rangle$ where E is a set of environment states, τ is a state transformer function, represented concisely, and $e_0 \in E$ is the initial state of the environment.

We now need to introduce a model of the agents that inhabit systems. Many architectures for agents have been reported in the literature [17], and one possibility would therefore be to directly use one of these models in our analysis. However, in order to ensure that our results are as general as possible, we choose to model agents simply as functions that map runs (assumed to end with an environment state) to actions (cf. [14, pp580–581]):

$$Ag : \mathcal{R} \rightarrow Ac$$

Notice that while environments are implicitly non-deterministic, agents are assumed to be deterministic.

We say a *system* is a pair containing an agent and an environment. Any system will have associated with it a set of possible runs; we denote the set of complete runs of agent Ag in environment Env by $\mathcal{R}(Ag, Env)$. Formally, a sequence $(e_0, \alpha_0, e_1, \alpha_1, e_2, \ldots)$ represents a run of an agent Ag in environment $Env = \langle E, \tau, e_0 \rangle$ iff

1. $e_0 = \tau(\epsilon)$ and $\alpha_0 = Ag(e_0)$ (where ϵ is the empty sequence); and
2. for $u > 0$,

$$\begin{array}{ll} e_u \in \tau((e_0, \alpha_0, \ldots, \alpha_{u-1})) & \text{where} \\ \alpha_u = Ag((e_0, \alpha_0, \ldots, e_u)) & \end{array}$$

3 Agent Design Tasks

We build agents in order to carry out *tasks* for us. We can identify many different *types* of tasks. The two most obvious of these are *achievement tasks* and *maintenance tasks*, as follows [16]:

1. Achievement tasks are tasks with the general form "achieve state of affairs φ".
2. Maintenance tasks are tasks with the general form "maintain state of affairs φ".

Intuitively, an achievement task is specified by a number of "goal states"; the agent is required to bring about one of these goal states. Note that we do not care *which* goal state is achieved — all are considered equally good. Achievement tasks are probably the most commonly studied form of task in artificial intelligence. Many well-known AI problems (such as the towers of Hanoi) are instances of achievement tasks. An achievement task is specified by some subset $\mathcal{G}$ (for "good" or "goal") of environment states E. An agent is *successful* on a particular run if it is guaranteed to bring about one of the states $\mathcal{G}$, that is, if every run of the agent in the environment results in one of the states $\mathcal{G}$. We say an agent Ag succeeds in an environment Env if every run of the agent in that environment is successful. An agent thus succeeds in an environment if it can *guarantee* to bring about one of the goal states.

We refer to a tuple $\langle Env, \mathcal{G} \rangle$, where Env is an environment and $\mathcal{G} \subseteq E$ is a set of environment states as a *task environment*. We can identify the following agent design problem for achievement tasks:

> <u>ACHIEVEMENT AGENT DESIGN</u>
> *Given*: task environment $\langle Env, \mathcal{G} \rangle$.
> *Answer*: "Yes" if there exists an agent Ag that succeeds in $\langle Env, \mathcal{G} \rangle$, "No" otherwise.

This decision problem amounts to determining whether the following second-order logic formula is true, for a given task environment $\langle Env, \mathcal{G} \rangle$:

$$\exists Ag \cdot \forall r \in \mathcal{R}(Ag, Env) \cdot \exists e \in \mathcal{G} \cdot e \text{ occurs in } r.$$

We emphasise that achievement tasks are emphatically *not* simply graph search problems. Because the environment can be *history dependent*, the solution to an achievement design problem must be a strategy, which dictates not simply which action to perform for any given environment state, but which action to perform *for any given history*. Such strategies will be exponentially large in the size of $E \times Ac$.

Example 1. Consider the environment whose state transformer function is illustrated by the graph in Figure 1. In this environment, an agent has just four available actions (α_1 to α_4 respectively), and the environment can be in any of six states (e_0 to e_5). History dependence in this environment arises because the agent is not allowed to execute the same action twice. Arcs between states in Figure 1 are labelled with the actions that cause the state transitions — note that the environment is non-deterministic. Now consider the achievement problems determined by the following goal sets:

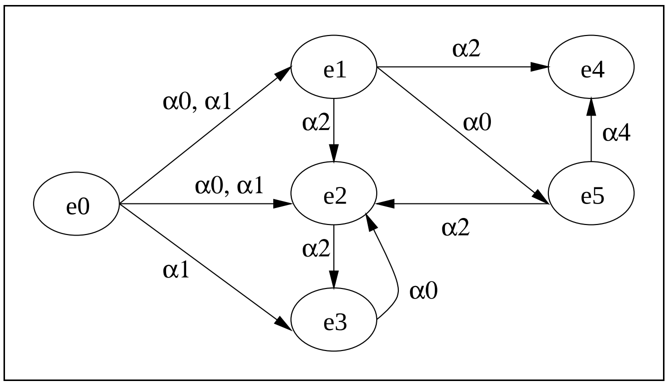

Fig. 1. The state transitions of an example environment: Arcs between environment states are labelled with the sets of actions corresponding to transitions. Note that this environment is *history dependent*, because agents are not allowed to perform the same action twice. So, for example, if the agent reached state e_2 by performing α_0 then α_2, it would not be able to perform α_2 again in order to reach e_3.

- $\mathcal{G}_1 = \{e_2\}$
  An agent can reliably achieve $\mathcal{G}_1$ by performing α_1, the result of which will be either e_1, e_2, or e_3. If e_1 results, the agent can perform α_0 to take it to e_5 and then α_2 to take it to e_2. If e_3 results, it can simply perform α_0.
- $\mathcal{G}_2 = \{e_3\}$
  There is no agent that can be guaranteed to achieve $\mathcal{G}_2$. If the agent performs α_1, then any of e_1 to e_3 might result. In particular, if e_1 results, the agent can only get to e_3 by performing α_2 twice, which is not allowed.

A useful way to think about ACHIEVEMENT AGENT DESIGN is as the agent *playing a game* against the environment. In the terminology of game theory [2], this is exactly what is meant by a “game against nature”. The environment and agent both begin in some state; the agent takes a turn by executing an action, and the environment responds with some state; the agent then takes another turn, and so on. The agent “wins” if it can *force* the environment into one of the goal states $\mathcal{G}$. The achievement design problem can then be understood as asking whether or not there is a *winning strategy* that can be played against the environment *Env* to bring about one of $\mathcal{G}$. This class of problem — determining whether or not there is a winning strategy for one player in a particular two-player game — is closely associated with PSPACE-complete problems [12, pp459–474].

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, that is, to maintain some invariant condition. 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 task environments as *maintenance* task environments.

A maintenance task environment is formally defined by a pair $\langle Env, \mathcal{B} \rangle$, where *Env* is an environment, and $\mathcal{B} \subseteq E$ is a subset of E that we refer to as the "bad", or "failure" states — these are the environment states that the agent must avoid. An agent is successful with respect to a maintenance task environment $\langle Env, \mathcal{B} \rangle$ if no state in $\mathcal{B}$ occurs on any run in $\mathcal{R}(Ag, Env)$.

Example 2. Consider again the environment in Figure 1, and the maintenance tasks defined by the following bad sets:

– $\mathcal{B}_1 = \{e_5\}$
  There is clearly an agent that can avoid e_5. After the agent performs its first action (either α_0 or α_1), one of the three states e_1 to e_3 will result.
  If the state that results is e_1, then the agent can perform α_2, after which either e_4 or e_2 will result; there will be no allowable moves in either case.
  If the state that results is e_2, then the agent can only perform α_2, which will transform the environment to e_4. The only allowable move will then be α_0 (if this has not already been performed — if it has, then there are no allowable moves); if the agent performs α_0, then environment state e_2 will result, from where there will be no allowable moves.
  Finally, if the state that results is e_3, then the agent can only perform α_0 and then α_2, which returns the environment to state e_3 from where there are no allowable moves.
– $\mathcal{B}_1 = \{e_2\}$
  No agent can be guaranteed to avoid e_2. Whether or not the first action is α_0 or α_1, it is possible that e_2 will result.

Given a maintenance task environment $\langle Env, \mathcal{B} \rangle$, the MAINTENANCE AGENT DESIGN decision problem simply involves determining whether or not there exists some agent that succeeds in $\langle Env, \mathcal{B} \rangle$. It is again useful to think of MAINTENANCE AGENT DESIGN as a game. This time, the agent wins if it manages to *avoid* $\mathcal{B}$. The environment, in the role of opponent, is attempting to force the agent into $\mathcal{B}$; the agent is successful if it has a winning strategy for avoiding $\mathcal{B}$. It is not hard to see that the MAINTENANCE AGENT DESIGN problem for a given $\langle Env, \mathcal{B} \rangle$ amounts to determining whether or not the following second-order logic formula is true:

$$\exists Ag \cdot \forall r \in \mathcal{R}(Ag, Env) \cdot \forall e \in \mathcal{B} \cdot e \text{ does not occur in } r.$$

Intuition suggests that MAINTENANCE AGENT DESIGN must be *harder* that ACHIEVEMENT AGENT DESIGN. This is because with achievement tasks, the agent is only required to bring about $\mathcal{G}$ *once*, whereas with maintenance tasks environments, the agent must avoid $\mathcal{B}$ *indefinitely*. However, this turns out not to be the case.

Theorem 1 (From [16]). *Both* ACHIEVEMENT AGENT DESIGN *and* MAINTENANCE AGENT DESIGN *are* PSPACE*-complete in the most general case (where environments may be history dependent).*

We remark that although the precise relationship of PSPACE-complete problems to, for example, NP-complete problems is not (yet) known, it is generally believed that they are much harder in practice. For this reason, PSPACE-completeness results are interpreted as much more negative than "mere" NP-completeness.

4 Optimistic Agent Design

In this paper, we focus on some variations of ACHIEVEMENT AGENT DESIGN and MAINTENANCE AGENT DESIGN. These variations allow us to consider weaker requirements for agent design, and also progressively more complex types of tasks for agents. The first variation we consider is *optimistic* agent design (cf. [8]). The intuition is that, in our current agent design problems, we are looking for an agent that can be *guaranteed* to carry out the task. That is, the agent is required to succeed with the task on *every possible* run. This is a rather severe requirement: after all, when we carry out tasks in the real world, there is frequently some possibility — even if rather remote — that we will fail. We would still be inclined to say that we have the capability to carry out the task, even though we know that, in principle at least, it is possible that we will fail.

This consideration is our motivation for a more relaxed notion of agent design. If P denotes either the ACHIEVEMENT AGENT DESIGN or MAINTENANCE AGENT DESIGN, then by OPTIMISTIC P we mean the variant of this problem in which an agent is deemed to succeed if there is *at least one run* of the agent in the environment that succeeds with respect to the task. So, for example, an instance of OPTIMISTIC ACHIEVEMENT AGENT DESIGN (OAD) is given by a tuple $\langle Env, \mathcal{G} \rangle$, as with ACHIEVEMENT AGENT DESIGN. The goal of the problem is to determine whether or not there exists an agent Ag such that at least one member of $\mathcal{G}$ occurs on at least one run of Ag when placed in Env. Formally, an OAD problem can be understood as determining whether the following second-order logic formula is true:

$$\exists Ag \cdot \exists r \in \mathcal{R}(Ag, Env) \cdot \exists e \in \mathcal{G} \cdot e \text{ occurs in } r.$$

Notice the difference between the pattern of quantifiers in this expression and that for ACHIEVEMENT AGENT DESIGN.

Example 3. Consider again the environment in Figure 1, and the optimistic achievement design problems defined by the following sets:

– $\mathcal{G}_3 = \{e_1\}$
  Recall that no agent can *guarantee* to bring about e_1. However, there clearly exists an agent that can optimistically achieve e_1: the agent simply performs α_0.

Intuition tells us that optimistic variants of ACHIEVEMENT and MAINTENANCE AGENT DESIGN are easier than their regular variants, as we are proving an existential rather than a universal (one run rather than all runs). And for once, intuition turns out to be correct. We can prove the following.

Theorem 2. *Both* OAD *and* OMD *are* NP-*complete.*

Proof. *We do the proof for* OAD*: the* OMD *case is similar. We need to show that (i) the problem is in* NP*; and (ii) some known* NP*-complete problem can be reduced to* OAD *in polynomial time. Membership of* NP *is established by the following non-deterministic algorithm for* OAD*. Given an instance $\langle Env, \mathcal{G} \rangle$ of* OAD*, begin by guessing a run $r \in \mathcal{R}$ such that this run ends with a member of $\mathcal{G}$, and verify that the run is consistent with the state transformer function τ of Env. Since the length of the run will be at most polynomial in the size of $Ac \times E$, guessing and verifying can be done in non-deterministic polynomial time. Given such a run, extracting the corresponding agent is trivial, and can be done in (deterministic) polynomial time.*

To prove completeness, we reduce the DIRECTED HAMILTONIAN CYCLE (DHC) *problem to* OAD *[6, p199]:*

DIRECTED HAMILTONIAN CYCLE (DHC)*:*
Given*: A directed graph $G = (V, F \subseteq V \times V)$*
Answer*: "Yes" if G contains a directed Hamiltonian cycle, "No" otherwise.*

The idea of the reduction is to encode the graph G 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 $G = (V, F \subseteq V \times V)$ of DHC*, we generate an instance of* OAD *as follows. First, create the set of environment states as follows:*

$$E = V \cup \{succeed\}$$

We then define the initial state of the environment as follows:

$$e_0 = v_0$$

We create an action $\alpha_{i,j}$ corresponding to every arc in G:

$$Ac = \{\alpha_{i,j} \mid \langle v_i, v_j \rangle \in F\}$$

We define $\mathcal{G}$ to be a singleton:

$$\mathcal{G} = \{succeed\}$$

And finally, we define the state transformer function τ in two parts. The first case deals with the first action of the agent:

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

The second case deals with subsequent actions:

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

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 node. Since the construction is clearly polynomial time, we are done.

As an aside, note that this proof essentially involves interpreting the Hamiltonian cycle problem as a game between a deterministic, history dependent environment, and an agent. The objective of the game is to visit every vertex of the graph exactly once. The game is history dependent because the "player" is not allowed to revisit vertices.

5 Disjunctive Agent Design

The next variations of agent design that we consider involve *combining* achievement and maintenance problems. The idea is that we specify a task by means of *both* a set $\mathcal{G} \subseteq E$ *and* a set $\mathcal{B} \subseteq E$. A run will be said to satisfy such a task if it contains at least one state in $\mathcal{G}$, and contains no states in $\mathcal{B}$. As before, we say an agent *Ag* succeeds in an environment *Env* with such a task if every run of the agent in the environment satisfies the task, i.e., if every run contains at least one state in $\mathcal{G}$ and no states in $\mathcal{B}$. Note that we do not require the *same* states to be achieved on different runs — all states in $\mathcal{G}$ are considered equally good.

Example 4. With respect to the environment in Figure 1, consider the following combined design tasks:

– $\mathcal{G}_4 = \{e_2\}$ and $\mathcal{B}_3 = \{e_4\}$.
  There is clearly an agent that can be guaranteed to succeed with this task. The agent simply uses the strategy described above for the achievement task of e_2; this strategy avoids e_4.
– $\mathcal{G}_5 = \{e_2\}$ and $\mathcal{B}_4 = \{e_5\}$.
  There is no agent that can be guaranteed to bring about e_2 while avoiding e_5.

We refer to a triple $\langle Env, \mathcal{G}, \mathcal{B}\rangle$, where *Env* is an environment, and $\mathcal{G}, \mathcal{B} \subseteq E$, as a *combined task environment*. Obviously, if $\mathcal{G} \subseteq \mathcal{B}$, then no agent will exist to carry out the task.

It turns out that the COMBINED AGENT DESIGN problem is in fact no harder than either ACHIEVEMENT AGENT DESIGN or MAINTENANCE AGENT DESIGN. (The problem is of course no easier, as any ACHIEVEMENT AGENT DESIGN problem $\langle Env, \mathcal{G}\rangle$ can trivially be reduced to a COMBINED AGENT DESIGN problem $\langle Env, \mathcal{G}, \emptyset\rangle$.) We will see that it is in fact a special case of a yet more general type of problem, called *disjunctive* agent design. The idea in a disjunctive task is that we give an agent a number of *alternative* goals to achieve, where each goal is associated with a corresponding invariant condition. An agent is successful with such a task if, on every possible run, it brings about one of the goals without invalidating the corresponding invariance.

To make this more precise, a disjunctive agent design problem is specified using a set of pairs with the form:

$$\{\langle \mathcal{G}_1, \mathcal{B}_1\rangle, \ldots, \langle \mathcal{G}_n, \mathcal{B}_n\rangle\}$$

Here, $\mathcal{G}_i$ is a set of goal states and $\mathcal{B}_i$ is the invariant corresponding to $\mathcal{G}_i$. A run will be said to satisfy such a task if every run satisfies at least one of the pairs $\langle \mathcal{G}_i, \mathcal{B}_i\rangle \in \{\langle \mathcal{G}_1, \mathcal{B}_1\rangle, \ldots, \langle \mathcal{G}_n, \mathcal{B}_n\rangle\}$. In other words, a run r satisfies task $\{\langle \mathcal{G}_1, \mathcal{B}_1\rangle, \ldots, \langle \mathcal{G}_n, \mathcal{B}_n\rangle\}$ if it satisfies $\langle \mathcal{G}_1, \mathcal{B}_1\rangle$ or it satisfies $\langle \mathcal{G}_2, \mathcal{B}_2\rangle$ or ... or it satisfies $\langle \mathcal{G}_n, \mathcal{B}_n\rangle$.

Formally, this problem involves asking whether the following second-order formula is true:

$$\exists Ag \cdot \forall r \in \mathcal{R}(Ag, Env) \cdot \exists i \cdot [(\exists e \in \mathcal{G}_i \cdot e \text{ is in } r) \text{ and } (\forall e \in \mathcal{B}_i \cdot e \text{ is not in } r)]$$

Notice the following subtleties of this definition:

– An agent is *not* required to bring about the *same* goal on each run in order to be considered to have succeeded. Different goals on different runs are perfectly acceptable.
– If an agent brings about some state in $\mathcal{G}_i$ on a run r and no state in r is in $\mathcal{B}_i$, then the fact that some states in $\mathcal{B}_j$ occur in r (for $i \neq j$) is not relevant — the agent is still deemed to have succeeded on run r.

We can prove the following.

Theorem 3. DISJUNCTIVE AGENT DESIGN *is* PSPACE*-complete.*

Proof. As before, we need to establish that DISJUNCTIVE AGENT DESIGN *(i) is in* PSPACE*, and (ii) is* PSPACE*-hard.* PSPACE*-hardness follows immediately from Theorem 1: any instance of* ACHIEVEMENT AGENT DESIGN *or* MAINTENANCE AGENT DESIGN *can be immediately reduced to an instance of* DISJUNCTIVE AGENT DESIGN*. We therefore focus on establishing membership of* PSPACE.

We give the design of a non-deterministic polynomial space Turing machine M that accepts instances of the problem that have a successful outcome, and rejects all others. The inputs to the algorithm will be a task environment $\langle Env, \{\langle \mathcal{G}_1, \mathcal{B}_1\rangle, \ldots, \langle \mathcal{G}_n, \mathcal{B}_n\rangle\}\rangle$ together with a run $r = (e_0, \alpha_0, \ldots, \alpha_{k-1}, e_k)$ — the algorithm actually decides whether or not there is an agent that will succeed in the environment given this current run. Initially, the run r will be set to the empty sequence ϵ. The algorithm for M is as follows:

1. *if r ends with an environment state in $\mathcal{G}_i$, for some $1 \leq i \leq n$, then check whether any member of $\mathcal{B}_i$ occurs in r — if the answer is no, then M accepts;*
2. *if there are no allowable actions given r, then M rejects;*
3. *non-deterministically choose an action $\alpha \in Ac$, and then for each $e \in \tau(r \cdot \alpha)$ recursively call M with the run $r \cdot \alpha \cdot e$;*
4. *if all of these accept, then M accepts, otherwise M rejects.*

The algorithm thus non-deterministically explores the space of all possible agents. Notice that since any run will be at most polynomial in the size of $E \times Ac$, the depth of recursion stack will be also be at most polynomial in the size of $E \times Ac$. Hence M requires only polynomial space. Hence DISJUNCTIVE AGENT DESIGN *is in non-deterministic polynomial space (*NPSPACE*). But since* PSPACE = NPSPACE *[12, p150], it follows that* DISJUNCTIVE AGENT DESIGN *is also in* PSPACE.

Note that in DISJUNCTIVE AGENT DESIGN, we require an agent that, on every run, both achieves some state in $\mathcal{G}_i$ while avoiding all states in $\mathcal{B}_i$. We can consider a variant of DISJUNCTIVE AGENT DESIGN, as follows. The idea is that on every run, an agent must *either* achieve some state in $\mathcal{G}_i$, *or* avoid all states in $\mathcal{B}_i$. This condition is given by the following second-order formula:

$$\exists Ag \cdot \forall r \in \mathcal{R}(Ag, Env) \cdot \exists i \cdot [(\exists e \in \mathcal{G}_i \cdot e \text{ is in } r) \text{ or } (\forall e \in \mathcal{B}_i \cdot e \text{ is not in } r)]$$

We refer to this problem as WEAK DISJUNCTIVE AGENT DESIGN. It is not difficult to prove that this problem is also PSPACE-complete.

Theorem 4. WEAK DISJUNCTIVE AGENT DESIGN *is* PSPACE-*complete.*

Proof. PSPACE-*hardness follows from Theorem 1. To show membership, we give the design of a non-deterministic polynomial space Turing machine M that will decide the problem — the idea is similar to Theorem 3:*

1. *if r ends with an environment state in $\mathcal{G}_i$, for some $1 \leq i \leq n$, then M accepts;*
2. *if there are no allowable actions given r, and there is some $1 \leq i \leq n$ such that no element of $\mathcal{B}_i$ occurs on r, then M accepts;*
3. *if there are no allowable actions given r, then M rejects;*
4. *non-deterministically choose an action $\alpha \in Ac$, and then for each $e \in \tau(r \cdot \alpha)$ recursively call M with the run $r \cdot \alpha \cdot e$;*
5. *if all of these accept, then M accepts, otherwise M rejects.*

The remainder of the proof is as Theorem 3.

6 Related Work

Probably the most relevant work from mainstream computer science to that discussed in this paper has been on the application of temporal logic to reasoning about systems [9, 10]. Temporal logic has been particularly applied to the specification of *non-terminating* systems. Temporal logic is particularly appropriate for the specification of such systems because it allows a designer to succinctly express complex properties of infinite sequences of states.

We identified several decision problems for agent design, and closely related problems have also been studied in the computer science literature. Perhaps the closest to our view is the work of Manna, Pnueli, Wolper, and colleagues on the automatic synthesis of systems from temporal logic specifications [4, 11, 13]. In this work, tasks are specified as formulae of temporal logic, and constructive proof methods for temporal logic are used to construct model-like structures for these formulae, from which the desired system can then be extracted.

In artificial intelligence, the planning problem is most closely related to our achievement tasks [1]. Bylander was probably the first to undertake a systematic study of the complexity of the planning problem; he showed that the (propositional) STRIPS decision problem is PSPACE-complete [3]. Building on his work, many other variants of the planning problem have been studied — a recent example is [8]. The main differences between our work and this are as follows:

– Most complexity results in the planning literature assume *declarative specifications* of goals and actions — the STRIPS representation is commonly used. In some cases, it is therefore not clear whether the results obtained reflect the complexity of the decision task, or whether they are an artifact of the chosen representation. Some researchers, noting this, have considered “flat” representations, where, as in our work, goals are specified as sets of states, rather than as logical formulae [8].
– Most researchers have ignored the properties of the environment; in particular, we are aware of no work that considers history dependence. Additionally, most research assumes deterministic environments.
– As far as we are aware, no other research in AI planning has considered complex task specifications — the focus appears to be almost exclusively on achievement goals.

Recently, there has been renewed interest by the artificial intelligence planning community in *Partially Observable Markov Decision Processes* (POMDPs) [7]. Put simply, the goal of solving a POMDP is to determine an optimal policy for acting in an environment in which there is uncertainty about the environment state, and which is non-deterministic. Finding an optimal policy for a POMDP problem is similar to our agent design problem.

7 Conclusions

In this paper, we have investigated the computational complexity of two important classes of agent design problem. In a combined agent design problem, we define a task via a set of goal states and a set of states corresponding to an invariant; we ask whether there exists an agent that can be guaranteed to achieve the goal while maintaining the invariant. In a disjunctive agent design problem, a task is specified by a set of goals and corresponding invariants; we ask whether there exists an agent that will bring about a goal state while maintaining the corresponding invariant. We have demonstrated that both of these problems are PSPACE-complete, and are hence no worse than the (intuitively simpler) achievement and maintenance problems discussed in [16].

We are currently investigating one obvious generalisation of agent design problems, to allow for arbitrary boolean combinations of tasks, rather than simple disjunctions. Cases where the PSPACE-completeness results collapse to NP or P are also being investigated. Finally, multi-agent variants seem worthy of study (cf. [15]).

Acknowledgments: We thank the reviewers for their detailed, insightful, and extremely helpful comments.

References

1. J. F. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann Publishers: San Mateo, CA, 1990.
2. K. Binmore. *Fun and Games: A Text on Game Theory*. D. C. Heath and Company: Lexington, MA, 1992.

3. T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.
4. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs — Proceedings 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Berlin, Germany, 1981.
5. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of* NP*-Completeness*. W. H. Freeman: New York, 1979.
7. L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.
8. M. L. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of AI Research*, 9:1–36, 1998.
9. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.
10. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.
11. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.
12. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley: Reading, MA, 1994.
13. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on the Principles of Programming Languages (POPL)*, pages 179–190, January 1989.
14. S. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of AI Research*, 2:575–609, 1995.
15. M. Tennenholtz and Y. Moses. On cooperation in a multi-entity model: Preliminary report. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, 1989.
16. M. Wooldridge. The computational complexity of agent design problems. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, Boston, MA, 2000.
17. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.