The complexity of achievement and maintenance problems in agent-based systems

Iain A. Stewart*,
Department of Computer Science,
University of Durham, Durham DH1 3LE, U.K.

Abstract

We completely classify the computational complexity of the basic achievement and maintenance agent design problems in bounded environments when these problems are parameterized by the number of environment states and the number of agent actions. The different problems are **P**-complete, **NP**-complete, **co-NP**-complete or **PSPACE**-complete (when they are not trivial). We also consider alternative achievement and maintenance agent design problems by allowing longer runs in environments (that is, our environments are bounded but the bounds are more liberal than was the case previously). Again, we obtain a complete classification but so that the different problems are **DEXPTIME**-complete, **NEXPTIME**-complete, **co-NEXPTIME**-complete or **NEXPSPACE**-complete (when they are not trivial).

1 Introduction

Agent-based systems have been the subject of much research in artificial intelligence and computer science (see, for example, [21]). However, the computational complexity of basic agent-environment problems has not been seriously considered; although, it has to be said, a number of similar and related problems in areas such as planning, model checking and Petri nets have been extensively studied. Examples from planning include: the basic planning problem, that is, the problem of deciding whether a given instance of the propositional STRIPS planning problem has a solution, *i.e.*, a plan, which was proven to be **PSPACE**-complete [4]; a whole range of planning problems in probabilistic planning domains [15]; planning problems in the presence of incompleteness [2]; and planning problems with temporal goals [3] (the reader is also referred to the survey paper [5]). Examples from model checking and Petri nets include: the problem of model checking with respect to Kripke structures and linear temporal logic, LTL, which was proven to be **PSPACE**-complete [17]; and the problems of model checking with respect to 1-safe Petri nets and both LTL and computation tree logic, CTL, which was proven to be **PSPACE**-complete [13] (the reader is referred to the survey papers [9] and [10]).

*Most of this work was completed whilst the author was at the University of Leicester.

The only complexity-theoretic considerations of agent-environment problems so far are those due to Dunne and Wooldridge who study: the basic achievement and maintenance agent design problems [18]; more sophisticated achievement and maintenance agent design problems [20, 8]; and the basic agent verification problem [7]. As might be expected (given complexity results from planning, model checking and Petri nets), these basic agent-environment problems turn out to be **PSPACE**-complete. The essential differences between agent-environment problems and many similar problems in, for example, planning, model checking and Petri nets are that the evolution of an agent-environment interaction is history-dependent (and not just state-dependent) and the environment can behave non-deterministically. Agent-environment interactions can be interpreted as games between two players where one player, the agent, is usually attempting to achieve or maintain some property whilst the other player, the environment, is attempting to make life as difficult as possible for the agent. Basic agent-environment problems amount to asking the question of whether the agent has a winning strategy for the particular game.

In this paper, we refine the analysis in [18] so as to completely classify the achievement and maintenance agent design problems when these problems are parameterized by the number of environment states and the number of agent actions (the constructions in [18] are not sophisticated enough for us to be able to ascertain these results). In order to exhibit our classification, we work with alternating Turing machines (ATMs) and derive complexity-theoretic completeness results from first principles (that is, instead of working with known complete problems for some complexity class, we encode 'raw' machine computations as instances of our problems). Focussing on ATMs allows us to apply control to our parameters and is entirely natural. By this latter remark, we mean that the fact that states of an ATM can alternate between 'existential' states and 'universal' states enables us to encode ATM computations as agent-environment interactions (or games). It is generally the case that 'first principles' proofs are rare in the related literature in planning, model checking and Petri nets but given our wish to focus on additional parameters, they appear to be necessary in our circumstances.

Our basic achievement agent design problem is formulated using the notion of a bounded environment. Intuitively, such bounded environments correspond to an agent having to achieve some state of affairs 'quickly'. One can also consider maintenance agent design in such bounded environments but a more relevant formulation of the maintenance agent design problem is in a bounded environment where the bound is interpreted more liberally, so that agents must sustain some state of affairs for a 'longer' period (having an agent sustain some state of affairs indefinitely leads to undecidable problems). Our reasoning here is that maintenance problems tend to be concerned with ensuring 'something doesn't happen' over a significantly long period. We prove that restricting the number of states and actions in agent-based systems leads: to achievement agent design problems that are **P**-complete, **NP**-complete, **co-NP**-complete and **PSPACE**-complete; and to maintenance agent design problems (where the bound on the length of runs within an environment is exponentially longer than in the achievement agent design problem) that are **DEXPTIME**-complete, **NEXPTIME**-complete, **co-NEXPTIME**-complete and **NEXPSPACE**-complete.

Our results should be of some interest to practitioners as they show that even if we drastically reduce the numbers of states and actions in agent-based systems, most

of the basic achievement and maintenance design problems remain (very) intractable. As regards complexity-theoretic bounds such as those mentioned above, it should be noted that the fact that a problem is **NP**-complete essentially means that general instances of the problem cannot be solved in practice, even when the size of the instance is not particularly large. Nevertheless, there do exist heuristic methods for certain **NP**-complete problems (such as SAT solvers) which cope pretty well with quite sizeable instances arising in practice. However, a problem that is **PSPACE**-complete, **NEXPTIME**-complete or **NEXPSPACE**-complete can be interpreted as being 'exponentially harder' than **NP**-complete problems and, as such, out of the reach of such heuristic methods.

2 Agents and Environments

In this section, we detail a formal model for the analysis of environments and agents (the reader is referred to [19] for additional details). Essentially, we deal with finite-state systems consisting of an environment and an agent, whereby the agent interacts with the environment by performing actions, and the resulting actions change the state of the environment.

An *environment* $\mathcal{ENV}$ is given by:

- a finite set E of *states* and an *initial state* $e_0 \in E$;
- a finite set A of *actions*; and
- a *state transformer function* τ presented in the form of a polynomial-time deterministic Turing machine.

(In practice, we might think of both E and A as finite initial segments of natural numbers, with e_0 as 0.) The *size* of an environment $\mathcal{ENV} = (e_0, E, A, \tau)$ is $|E| + |A| + |\tau|$, where $|\tau|$ is the length of a reasonable encoding of the given Turing machine τ (we often use τ to denote the state transformer function and a polynomial-time deterministic Turing machine computing this function). It will always be the case that any environment is in a specific state (from E), and depending upon which action is chosen by an agent and also the history of the interaction between the environment and the agent, the Turing machine τ will compute the set of possible next states of the environment. Hence, τ can be viewed as (the description of) a polynomial-time computable function which takes an interaction history (that is, a sequence of states and actions, starting with the state e_0, alternating between a state and an action, and ending with a state) and an action as input and yields a set of states as output. Associated with an environment is the set of *potential runs* P defined as (the regular set) $\{e_0\}(AE)^*$ (where A and E are the sets of actions and states), *i.e.*, the set of finite tuples whose first component is e_0 and thereafter components alternate between actions and states so that the final component is a state. The *length* of any potential run is the number of actions therein, *i.e.*, if the tuple has $2n + 1$ components then the length of the tuple is n. However, not every potential run will be a legitimate run: the legitimate runs will only be those permitted according to the interaction of an agent with the environment.

An *agent* $\mathcal{AG}$ acts within an environment according to a given *agent strategy function* $\sigma : P \rightarrow A \cup \{\epsilon\}$, where ϵ is a new symbol hitherto unused. The set R of *legitimate runs* of the *agent-based system* $\langle \mathcal{ENV}, \mathcal{AG} \rangle$ is defined inductively as follows:

- (e_0) is a legitimate run; and
- if ρ is a legitimate run, $\sigma(\rho) = a \in A$ and $e \in \tau(\rho, a)$ then (ρ, a, e) is a legitimate run

(where (ρ, a, e) denotes the concatenation of a and e onto the tuple ρ). Note how τ is used as a function which, given a legitimate run and an action, computes the set of possible next states into which the environment can evolve (and so the state-transformer function is history-dependent). Note also that whilst the agent behaves deterministically, the environment can behave non-deterministically. We call a legitimate run r for which either $\sigma(\rho) = \epsilon$ or $(\sigma(\rho) = a$ and $\tau(\rho, a) = \emptyset)$ a *complete* legitimate run (that is, a legitimate run which can not be further extended in the given system). We shall only ever be interested in the legitimate runs within any system $\langle \mathcal{ENV}, \mathcal{AG} \rangle$.

Agent-based systems are intended to model real systems. Hence, it is often the case that we are not just looking for some particular circumstance in our system but whether this circumstance has come about within some given time. For example, amongst the sort of problems that have hitherto been considered of agent-based systems are 'achievement' problems, where it is required that an agent achieves some specific state of affairs, and 'maintenance' problems, where it is required that an agent maintains a specific state of affairs. With achievement problems it is usually the case that the state of affairs in hand should be achieved within some reasonable time, *e.g.*, a search for data over the Internet should quickly register whether the data has been found or not. However, with maintenance problems it is usually the case that the state of affairs should be maintained in perpetuity, *e.g.*, a web-crawler should never be given access to data to which it does not have security clearance. Consequently, we also consider *bounded environments* which are environments where, as well as the set of states E (and the initial state $e_0 \in E$), the set of actions A and the state transformer function τ, a natural number b, the *bound*, is also supplied; with the result that the size of a bounded environment is $|E| + |A| + |\tau| + b$. In an agent-based system where the environment is bounded, the set of legitimate runs is defined as above except that now no legitimate run is allowed to contain more than b actions; that is, the new set of legitimate runs is obtained from the old set by retaining all legitimate runs of length at most b and truncating every run of length greater than b so that only the first b actions occur.

An example of an achievement problem, as was studied in [18], is the problem ACHIEVEMENT AGENT DESIGN for which: an instance is a pair $(\mathcal{ENV}, G)$, where $\mathcal{ENV} = (e_0, E, A, \tau, b)$ is a bounded environment and $G \subseteq E$ is a set of *goal states*; and a yes-instance is an instance for which there exists an agent $\mathcal{AG}$ such that every legitimate run contains a goal state. An example of a maintenance problem, again as studied in [18], is the problem MAINTENANCE AGENT DESIGN for which: an instance is a pair $(\mathcal{ENV}, B)$, where $\mathcal{ENV} = (e_0, E, A, \tau, b)$ is a bounded environment and $B \subseteq E$ is a set of *bad states*; and a yes-instance is an instance for which there exists an agent $\mathcal{AG}$ such that no legitimate run contains a bad state. Both

problems, ACHIEVEMENT AGENT DESIGN and MAINTENANCE AGENT DESIGN, were proven in [18] to be **PSPACE**-complete[1].

Note that even though the problem MAINTENANCE AGENT DESIGN is a maintenance problem, and so it makes sense to consider environments rather than bounded environments, the particular formulation in [18] was essentially for bounded environments. Indeed, if we were to relax the stipulation that an environment should be bounded then we would be faced with undecidable problems. Later on in this paper, we shall define an amended, more practically relevant version of the problem MAINTENANCE AGENT DESIGN where environments are still bounded but where runs in the environment can be exponentially longer.

Before we proceed, we have one remark to make. Consider the decision problem ACHIEVEMENT AGENT DESIGN. When instances are encoded for input to, say, a (deterministic) Turing machine, we should be able to easily ascertain whether an input string is the encoding of an instance. In particular, we should be able to check whether the Turing machine (within the encoding) is a polynomial-time Turing machine. Unfortunately, by Rice's Theorem (see, for example, [16]), it is undecidable to check whether a given Turing machine is a polynomial-time Turing machine. Hence, we encode our polynomial time Turing machine as a pair, the first component of which is the encoding of a Turing machine M, and the second component of which is an integer k; and we insist that any computation of M of length greater than $n^k + k$ (on an input of length n) halts as a rejecting computation. That is, (M, k) is essentially a 'clocked' Turing machine. Any polynomial-time Turing machine can be realised as a clocked Turing machine, and *vice versa*.

3 Alternating Turing machines

Alternating Turing machines were introduced in [6] and [14] as a model of parallel machines. However, they have proven to be very useful with regard to sequential complexity too. The reader is referred to [1] and [12] for an extensive discussion of alternating Turing machines and for any details omitted below.

An *alternating Turing machine* (*ATM*) $M = \langle Q, R, F, \Gamma, \delta \rangle$ is a non-deterministic Turing machine with a read-only *input-tape*, a read-write *index-tape* and one read-write *work-tape* (though there can be more work-tapes if needs be). All tapes are two-way infinite and on each tape, cells are indexed by the integers. The index-tape and the work-tape have associated *tape-heads*. The input string ω, over $\{0, 1\}$ and of length n, is presented on the input-tape in cells 1, 2, ..., n, with all other cells of the input-tape and every cell of the index- and work-tapes initially holding a special *blank symbol* $\sharp$. The finite *work-tape alphabet* Γ contains 0, 1 and $\sharp$ and may also contain other symbols. The symbols from $\{0, 1, \sharp\}$ will be the only symbols appearing on the index-tape, and whenever a bit of the input string ω is to be read, the longest contiguous string of 0s and 1s on the index-tape (starting at cell 1), known as the *index*, is taken as the binary representation of the name of the cell on the input-tape to be read (if cell 1 holds the blank symbol then the index is 0).

[1] In [18], an attempt was made to only consider environments within which all (legitimate) runs are complete and have length 'polynomial in the size of $|A| \times |E|$'. This notion does not actually make sense and our definition, of a bounded environment, is the appropriate formalism. Nevertheless, the proofs in [18] of **PSPACE**-completeness essentially still hold for our formalism of the problems.

The set Q is the finite set of *states* of M and contains the *initial state* q_0 and the *accept state* q_a (to obviate the need to consider trivial cases, we insist that $q_0 \neq q_a$). The set of states $R \subseteq Q$ is the set of *read states* of M; and the set of states F is the set of *universal* states of M, with the states of $Q \setminus F$ being the *existential* states (whether a state is universal or existential impacts on whether an input string is accepted by an ATM, as we shall see below). The relation

$$\delta \subseteq (Q \setminus \{q_a\} \times \{0, 1, \sharp\}^2 \times \Gamma) \times (Q \times \{0, 1, \sharp\} \times \Gamma \times \{-1, 0, 1\}^2)$$

describes the transitions of M as follows.

- If $((q, s_1, s_2, s_3), (q', s_2', s_3', h_2, h_3)) \in \delta$ and $q \in R$ then:
  - if M is in state q and the index has the value i with the ith cell of the input-tape holding the symbol s_1
  - then M can move into state q' and no other tape and head alterations are made.

  So, s_2, s_3, s_2', s_3', h_2 and h_3 are redundant in such tuples of δ.
- If $((q, s_1, s_2, s_3), (q', s_2', s_3', h_2, h_3)) \in \delta$ and $q \notin R$ then:
  - if M is in state q; the index-head is scanning the symbol s_2; and the work-head is scanning the symbol s_3
  - then M can move into state q'; write the symbol s_2' on the index-tape and move the index-head h_2 cells to the right; and write the symbol s_3' on the work-tape and move the work-head h_3 cells to the right (with '-1' meaning a move of one cell to the left).

  So, s_1 is redundant in such tuples of δ.

An ATM M computes in the usual way, to accept a set of strings over $\{0, 1\}$, except that its notion of acceptance differs considerably from that of a non-deterministic Turing machine.

Suppose that M halts on every input (this will always be the case for us). An input string ω gives rise (in the usual way) to a finite computation tree T where the nodes are instantaneous descriptions (IDs) of M and where the edges (all directed away from the root) describe single transitions of M on input ω. Every node is either *existential* or *universal*, depending upon the state in the ID. For any node u of T, denote by $child^T(u)$ the number of children of u. By imposing an order on the transitions described by δ, we can clearly talk about the first child of a node u, the second child of u, and so on. A *valuation* μ on T is a function taking an existential node u of T as an input and yielding an integer in $\{0, 1, 2, \ldots, child^T(u)\}$ as output. We *apply* a valuation μ to T at every existential node, working away from the root in a breadth-first fashion, until there are no more existential nodes to consider, as follows.

- If p is a path in T from the root to an existential node u then:
  - if $\mu(u) = i$

– then erase all sub-trees of T, pendant from u with roots the jth child of u, for every $j \in \{1, 2, \ldots, child^T(u)\} \setminus \{i\}$.

Applying a valuation μ to T yields a sub-tree $\mu(T)$ of T. If the sub-tree $\mu(T)$ is such that every leaf is an accepting ID, *i.e.*, the state of the ID is q_a, then the valuation μ is a *true valuation*. The input string ω is *accepted* by M if, and only if, there is a true valuation on the corresponding computation tree T. The *time taken* by M to accept ω is the minimal height of any sub-tree $\mu(T)$, where μ ranges over all true valuations. The *space used* by M to accept ω is the minimum over any sub-tree $\mu(T)$, where μ ranges over all true valuations, of the maximal size of an ID in $\mu(T)$.

Crucial to our analysis is the relationship between time-bounded ATMs and time- and space-bounded (deterministic and non-deterministic) Turing machines (note that a non-deterministic Turing machine is just an ATM where all states are existential). We denote the classes of languages accepted by deterministic and non-deterministic Turing machines in $O(s(n))$ space (resp. $O(t(n))$ time) by DSPACE$(s(n))$ and NSPACE$(s(n))$ (resp. DTIME$(t(n))$ and NTIME$(t(n))$). We denote the class of languages accepted by ATMs in simultaneous $O(t(n))$ time and $O(s(n))$ space by ATISP$(t(n), s(n))$.

Theorem 1 (see [12]) *For every space constructible functions $s(n)$ and $t(n)$, for which $s(n) = \Omega(\log(n))$, we have that*:

- $NSPACE(s(n)) \subseteq ATISP(s(n)^2, s(n))$;
- $ATISP(t(n), s(n)) \subseteq NSPACE(t(n)^2)$. □

We remind the reader also of *Savitch's Theorem*.

Theorem 2 (see [1]) *For every space constructible function $s(n)$ for which $s(n) = \Omega(\log(n))$, we have that $NSPACE(s(n)) \subseteq DSPACE(s(n)^2)$.* □

Finally, we mention that all of the completeness results in this paper are with respect to logspace reductions; that is, when we prove that every problem in some complexity class can be reduced to our complete problem, the actual reduction can not only be computed using polynomial-time but also using logspace.

4 Achievement problems

We beginning by reconsidering the problem ACHIEVEMENT AGENT DESIGN, first investigated in [18]. Whilst we re-prove the fact that this problem is **PSPACE**-complete, our proof is such that we obtain much more information about this problem and some of its close relations; in that our proof enables us to focus directly upon, and so restrict, the number of states and actions in our constructed bounded environments.

Theorem 3 *The problem ACHIEVEMENT AGENT DESIGN is* **PSPACE***-complete.*

Proof We begin by proving the **PSPACE**-hardness of our problem in hand. By Theorem 1, any problem in **PSPACE** can be accepted by a polynomial-time ATM. Let M be an ATM which runs in time $t(n)$, for some polynomial $t(n)$. By introducing extra 'dummy' states if necessary, we can assume that:

- every node in any computation tree of M has at most 2 children;
- in traversing any path in any computation tree of M, starting from the root, we alternate between existential and universal nodes, with the root being an existential node;
- all leaves in any computation tree of M are existential nodes.

The amended ATM (if indeed we need to amend our original Turing machine), which we also denote by M, runs in time $ct(n)$, for some constant c.

Consider some computation tree T of M, resulting from the input string ω of length n. We define a bounded environment $\mathcal{ENV} = (e_0, E, A, \tau, b)$ and a set of goal states $G \subseteq E$ corresponding to T. The state set E is the set $\{q_l, q_r, q_a\}$, with the initial state, e_0, being q_l. The set of actions A is $\{l, r\}$. Given some path p in T, starting from the root and leading to an existential node, we obtain a tuple $(e_0, a_1, e_1, a_2, \ldots, a_m, e_m)$, where each $e_i \in E$ and each $a_i \in A$, as follows. Starting from the root, traverse p and at the ith existential node on p, where $1 \leq i \leq m$, the node u_i say (the root is the 0th), define: e_i to be the state q_a, if u_i is an accepting node, otherwise the state q_l (resp. q_r) if u_i is the left-child (resp. right-child) of its parent; and, if $i \neq m$, a_{i+1} to be l (resp. r) if the next node on p is the left-child (resp. right-child) of u_i. We say that the tuple $(e_0, a_1, e_1, a_2, \ldots, a_m, e_m)$ is the *existential trace* of p.

On input a potential run $\rho = (e_0, a_1, e_1, a_2, \ldots, a_m, e_m)$ and an action a, the output from the function τ is defined as follows. We begin by checking that ρ is an existential trace in T. If it isn't then τ yields no output: otherwise, ρ is the existential trace of some path p in T (note that ρ can be the existential trace of at most one path in T) and we ascertain whether there is an edge in T from the final node, u say, of p to a left-most child, if $a = l$, and to a right-most child, if $a = r$. If no such child exists then τ yields no output: otherwise, such a child, v say, exists and the output from τ is defined to be the set of states associated with the children of the node v (there are at most 2 and there may be none). The function τ can clearly be computed in polynomial time (polynomial in m, to be precise): in fact, in $O(m)$ time, *i.e.*, linear-time. Our set of goal states $G \subseteq E$ is the set $\{q_a\}$; and our bound b is set at $ct(n)$.

Having constructed an instance $(\mathcal{ENV} = (e_0, E, A, \tau, b), G)$ of ACHIEVEMENT AGENT DESIGN, we need to ensure that the process of construction can be undertaken using logspace. That this is the case is trivial except for the description of the function τ (remember, τ is not given explicitly but is presented as a polynomial-time deterministic Turing machine which computes the function in hand[2]). Let M_0 be a deterministic Turing machine which takes as input the description of an ATM M (as described in the opening paragraph of this proof), an input string ω to M and a sequence $(e_0, a_1, e_1, a_2, \ldots, a_m, e_m) \in E(AE)^*$. The Turing machine M_0 simulates a computation of M on ω, where M is treated as a non-deterministic Turing machine and the sequence $(e_0, a_1, e_1, a_2, \ldots, a_m, e_m)$ dictates the choices made at any point of the computation. If such a simulation does not exist then M_0 rejects the input, otherwise M_0 computes the function τ. Note that M_0 is a fixed Turing machine. It is straightforward to devise a logspace algorithm which takes M and ω as input

[2]This point was completely overlooked in [18].

and outputs (using M_0) a polynomial-time (in fact, linear-time) deterministic Turing machine which computes τ.

Let $\mathcal{AG}$ be any agent, with agent strategy function σ, acting within the bounded environment $\mathcal{ENV}$. This agent yields a valuation on the tree T in the natural way.

- If p is a path in T from the root to an existential node u for which the existential trace is ρ, then:
  - if $\sigma(\rho) = l$ (resp. r) and u has a left-child (resp. right-child)
  - then erase the right (resp. left) branch of T descending from u (if there is one).

Alternatively, any valuation on T clearly yields an agent acting within the bounded environment $\mathcal{ENV}$. Moreover, there is an agent such that every legitimate run contains a goal state if, and only if, there is a true valuation on the tree T. Hence, the problem ACHIEVEMENT AGENT DESIGN is **PSPACE**-hard.

Now we show that our problem is in **PSPACE**. Let $(\mathcal{ENV} = (e_0, E, A, \tau, b), G)$ be an instance of our problem of size n. Consider the tree T, of depth at most $2b$, constructed as follows. Nodes are categorized as to their distance from the root, with the distance equating to the *level* of the node. The level of the root is 0. Every node on an even level is labelled as a *universal* node; and every node on an odd level as an *existential* node. Also, every node on an even level will be labelled with a state of E; and every node on an odd level with a symbol from A. The root is labelled e_0. Let u be a node on the (even) level $2m$, for some $m \in \{0, 1, \ldots, b-1\}$, where u's label is $e \in E$. Suppose that the path p from the root to the node u induces a $(2k+1)$-tuple $(e_0, a_0, e_1, a_1, \ldots, a_m, e_m)$ of labels (alternating between states of E and symbols of A and starting with e_0). The node u has $|A|$ children (on level $2k+1$) with these children labelled with the symbols from A. The child on level $2k+1$ labelled with $a \in A$ has $|\tau((e_0, a_0, e_1, a_1, \ldots, a_m, e_m), a)|$ children (on level $2k+2$) with these children labelled with the elements of $\tau((e_0, a_0, e_1, a_1, \ldots, a_m, e_m), a)$. Finally, the tree is pruned by: first, removing any sub-trees pendant from a node (on an even level) whose associated label is in G; and, second, removing any leaves on an odd level (and so all leaves are existential nodes and labelled with states of E).

Just as we have a valuation on the computation tree of an ATM, so we have the notion of a valuation on our tree T as constructed above. Essentially, a *valuation* μ is such that we retain exactly one pendant sub-tree (if there is one) from every existential node. Clearly, $(\mathcal{ENV} = (e_0, E, A, \tau, b), G)$ is a yes-instance of ACHIEVEMENT AGENT DESIGN if, and only if, there is a *true valuation* μ on T, *i.e.*, one such that every leaf of $\mu(T)$ is labelled with a state of G.

In order to ascertain whether there is a true valuation on T, we perform a 'non-deterministic' depth-first search on T. Our depth-first search is such that whenever an existential node is encountered, we non-deterministically guess a child to move to next; and when we back-track in our depth-first search, having guessed a child of an existential node, we never guess another child but keep on moving back up the tree to the (universal) parent node (from which we continue the depth-first search as normal). If our search returns to the root having ascertained that every leaf encountered was labelled with a state from G then we accept otherwise we reject. This clearly results in a polynomial-space non-deterministic algorithm, and so, by Theorem 2, a **PSPACE** algorithm, for the problem ACHIEVEMENT AGENT DESIGN. The result follows. □

The proof of Theorem 3 actually yields a much stronger result than that stated.

Corollary 4 *The problem ACHIEVEMENT AGENT DESIGN restricted to bounded environments with* 3 *states and* 2 *actions is* **PSPACE***-complete.* □

Our proof of Theorem 3 is superior to that in [18] because: first, as we have just seen in Corollary 4, it yields additional information (note that Corollary 4 is not derivable from the proof in [18]); second, as we shall see later, it enables us to prove complexity results about problems related to ACHIEVEMENT AGENT DESIGN; third, it exhibits a strong relationship between agent-based systems and alternation in the theory of computation; and, fourth, it clarifies key issues which were ignored in [18].

Corollary 4 gives us a significant insight into how we might impose restrictive conditions so as to make the problem ACHIEVEMENT AGENT DESIGN feasibly solvable. For brevity, we denote the problem ACHIEVEMENT AGENT DESIGN restricted to bounded environments where the set of states has size at most j and the set of actions has size at most k by AAD(j, k).

Theorem 5 *The problem AAD*$(2, 2)$ *is* **NP***-complete.*

Proof We begin by proving that AAD$(2, 2)$ is in **NP**. Let $(\mathcal{ENV} = (e_0, E, A, \tau, b), G)$ be an instance of size n. W.l.o.g., we may assume that $E = \{q, q_a\}$, $A = \{l, r\}$, $G = \{q_a\}$ and $e_0 = q$. Consider the following polynomial-time non-deterministic algorithm.

Guess $a_1 \in A$.
If $\tau((q), a_1) = \emptyset$ then reject.
If $\tau((q), a_1) = \{q_a\}$ then accept.
If $\tau((q), a_1) = \{q, q_a\}$ or $\{q\}$ then
  guess $a_2 \in A$.
  If $\tau((q, a_1, q), a_2) = \emptyset$ then reject.
  If $\tau((q, a_1, q), a_2) = \{q_a\}$ then accept.
  If $\tau((q, a_1, q), a_2) = \{q, q_a\}$ or $\{q\}$ then
    guess $a_3 \in A$.
    If $\tau((q, a_1, q, a_2, q), a_3) = \emptyset$ then reject.
    If $\tau((q, a_1, q, a_2, q), a_3) = \{q_a\}$ then accept.
    If $\tau((q, a_1, q, a_2, q), a_3) = \{q, q_a\}$ or $\{q\}$ then
      ...
      guess $a_b \in A$.
      If $\tau((q, a_1, q, \ldots, a_{b-1}, q), a_b) = \emptyset$ then reject.
      If $\tau((q, a_1, q, \ldots, a_{b-1}, q), a_b) = \{q_a\}$ then accept.
      If $\tau((q, a_1, q, \ldots, a_{b-1}, q), a_b) = \{q, q_a\}$ or $\{q\}$ then reject.

This algorithm clearly witnesses that our problem is in **NP**.

Let M be a non-deterministic Turing machine running in time $t(n)$, for some polynomial $t(n)$. By introducing extra 'dummy' states if necessary, we can assume that every node in any computation tree of M has at most 2 children. The amended Turing machine (if indeed we need to amend our original Turing machine), which we also denote by M, runs in time $ct(n)$, for some constant c.

Consider some computation tree T of M, resulting from the input string ω of length n. We define a bounded environment $\mathcal{ENV} = (e_0, E, A, \tau, b)$ and a set of goal states $G \subseteq E$ corresponding to T. The state set $E = \{q, q_a\}$, with the initial state, e_0, being q. The set of actions A is $\{l, r\}$. We associate a state with every node of T as follows: if the node is accepting then its associated state is q_a; otherwise it is q.

Given a potential run $\rho = (e_0, a_1, e_1, a_2, \ldots, a_m, e_m)$ and an action a, the output from the function τ is defined as follows. We begin by checking that there is a path p in T, starting from the root and obtained from the tuple $(a_1, a_2, \ldots, a_m)$ by interpreting the symbol l (resp. r) as meaning 'go to the left-child (resp. right-child)', such that the states of the nodes obtained by traversing p yield the tuple $(e_0, e_1, \ldots, e_m)$. If this path p does not exist then τ yields no output. Otherwise, p exists, and if $a = l$ (resp. $a = r$) then the output of τ is the state of the left-child (resp. right-child) of the final node of p, if it exists: if it does not exist then τ yields no output. The function τ can clearly be computed in polynomial time: in fact, in linear-time. Our set of goal states $G \subseteq E$ is the set $\{q_a\}$; and our bound b is set at $ct(n)$. As in the proof of Theorem 3, the instance $(\mathcal{ENV} = (e_0, E, A, \tau, b), G)$ can be constructed using logspace (including providing a polynomial-time deterministic Turing machine computing τ).

Let $\mathcal{AG}$ be any agent, with agent strategy function σ, acting within the bounded environment $\mathcal{ENV}$. This agent yields a path in T in the natural way; and by construction of $(\mathcal{ENV}, G)$, there is an agent such that every legitimate run contains a goal state if, and only if, M accepts the input ω. The result follows. □

Note that the algorithm in the proof of Theorem 5 is actually a polynomial-time non-deterministic algorithm to solve the problem AAD$(2, k)$, for any $k \geq 2$. Hence, we immediately obtain the following corollary.

Corollary 6 *For every $k \geq 2$, the problem $AAD(2, k)$ is* **NP***-complete.* □

We now restrict the set of actions so that there is (essentially) only one possible agent strategy function.

Theorem 7 *The problem $AAD(3, 1)$ is* **co-NP***-complete.*

Proof We first remark that it is straightforward to see that a problem is in **co-NP** if, and only if, there exists a polynomial-time non-deterministic Turing machine M such that:

- the computation tree of M corresponding to some yes-instance of the problem is such that every leaf is accepting; and
- the computation tree of M corresponding to some no-instance of the problem is such there exists a leaf which is not accepting.

We say that the language accepted by M according to the above criteria is the language *co-accepted* by M. Hence, **co-NP** consists of those languages co-accepted by polynomial-time non-deterministic Turing machines.

We begin by proving that the problem AAD$(3, 1)$ is **co-NP**-hard. Let M be a non-deterministic Turing machine which runs in time $t(n)$, for some polynomial $t(n)$. By introducing extra 'dummy' states if necessary, we can assume that every node in

any computation tree of M has at most 2 children. The amended ATM (if indeed we need to amend our original Turing machine), which we also denote by M, runs in time $ct(n)$, for some constant c.

Consider some computation tree T of M, resulting from the input string ω of length n. We define a bounded environment $\mathcal{ENV} = (e_0, E, A, \tau, b)$ and a set of goal states $G \subseteq E$ corresponding to T. The state set E is the set $\{q_l, q_r, q_a\}$, with the initial state, e_0, being q_l. The set of actions A is $\{a\}$. Given some path p in T, starting from the root, we obtain a tuple $(e_0, a, e_1, a, \ldots, a, e_m)$, where each $e_i \in E$, as follows. Starting from the root, traverse p and at the ith node on p, where $1 \leq i \leq m$, the node u_i say (the root is the 0th), define: e_i to be the state q_a, if u_i is an accepting node, otherwise the state q_l (resp. q_r) if u_i is the left-child (resp. right-child) of its parent. We say that the tuple $(e_0, a_1, e_1, a_2, \ldots, a_m, e_m)$ is the *trace* of p.

On input a potential run $\rho = (e_0, a, e_1, a, \ldots, a, e_m)$ and the action a, the output from the function τ is defined as follows. We begin by checking that ρ is a trace in T. If it isn't then τ yields no output: otherwise, ρ is the trace of some path p in T (note that ρ can be the trace of at most one path in T). If ρ is the trace of some path p and the final node on p has no children then τ yields no output: otherwise, τ yields the set of states associated with the children of this final node (there are at most 2 and there may be none). The function τ can clearly be computed in polynomial time (polynomial in m, to be precise): in fact, in linear-time. Our set of goal states $G \subseteq E$ is the set $\{q_a\}$; and our bound b is set at $ct(n)$. As in the proof of Theorem 3, this construction can be completed in logspace (including providing a polynomial-time deterministic Turing machine computing τ).

Let $\mathcal{AG}$ be the agent whose agent strategy function σ yields the action a for every legitimate run. Note that if $\mathcal{AG}$ does not witness that $(\mathcal{ENV}, G)$ is a yes-instance of AAD$(3, 1)$ then no agent does. By construction, our reduction is such that M co-accepts a string ω if, and only if, every legitimate run in the agent-based system $\langle \mathcal{ENV}, \mathcal{AG} \rangle$ contains a goal state. That is, AAD$(3, 1)$ is **co-NP**-hard.

That AAD$(3, 1)$ is in **co-NP** is straightforward as verifying that an instance $(\mathcal{ENV} = (e_0, E, A, \tau, b), G)$ of AAD$(3, 1)$ is a yes-instance amounts to building a tree (of depth at most b for which every node has at most 3 sons) described by τ, whose nodes are labelled with states of E, and checking that every leaf is labelled by a node of G. □

The fact that AAD$(j, 1)$ is in **co-NP**, for any $j \geq 3$, is trivial, given the above proof; and so we obtain the following corollary.

Corollary 8 *For $j \geq 3$, the problem AAD$(j, 1)$ is* **co-NP***-complete.* □

All that remains is to consider the problem AAD$(2, 1)$ (as the problem AAD$(1, k)$ is trivial, for any $k \geq 1$). However, the proof of Theorem 7 obviates the need for much more analysis.

Corollary 9 *The problem AAD$(2, 1)$ is* **P***-complete.*

Proof Let the Turing machine M in the proof of Theorem 7 be deterministic. Then proceeding in that proof but with a state set $E = \{q_l, q_a\}$ yields that the problem AAD$(2, 1)$ is **P**-hard. The fact that AAD$(2, 1)$ is in **P** is straightforward. □

5 Maintenance problems

We begin my analysing the MAINTENANCE AGENT DESIGN problem, henceforth abbreviated to MAD, as we did the ACHIEVEMENT AGENT DESIGN. We adopt an analogous notation when referring to the various restrictions of the problem MAD. As it turns out, our 'first principles' approach from the previous section has accounted for most of the hard work.

Corollary 10 (*a*) *If* $j \geq 3$ *and* $k \geq 2$ *then MAD*(j,k) *is* **PSPACE***-complete.*

(*b*) *If* $j = 2$ *and* $k \geq 2$ *then MAD*(j,k) *is* **NP***-complete.*

(*c*) *If* $j \geq 3$ *and* $k = 1$ *then MAD*(j,k) *is* **co-NP***-complete.*

(*d*) *If* $j = 2$ *and* $k = 1$ *then MAD*(j,k) *is* **P***-complete.*

(*e*) *If* $j = 1$ *and* $k \geq 1$ *then MAD*(j,k) *is trivial.*

Proof (*a*) With respect to the proof of Theorem 3, w.l.o.g. we may further assume that every path in any computation tree of M ends in either an accepting node or a rejecting node, where we have designated a particular state of M to be the reject state. In the proof of Theorem 3, label the nodes of T so that: a node is labelled q_a if it is a rejecting node (formerly, this was the case if it was an accepting node); otherwise, it is labelled q_l or q_r (depending upon the child-parent relationship). The amended proof then yields that the problem MAD$(3,2)$ is **PSPACE**-hard. Our **PSPACE** algorithm to solve MAD$(3,2)$ is as was the **PSPACE** algorithm in the proof of Theorem 3 except that the depth-first search checks that no node is labelled by a bad state (as opposed to, formerly, that every leaf is labelled with a state of G). Hence, MAD$(3,2)$ is **PSPACE**-complete; and therefore so is MAD$(j,2)$, for all $j \geq 3$.

(*b*) With respect to the proof of Theorem 5, we can devise a similar algorithm, to the algorithm in that proof, to solve MAD$(2,2)$ where, at each stage: we reject if $\tau((q),a_i) = \{q_a\}$ or $\{q,q_a\}$; we accept if $\tau((q),a_i) = \emptyset$; and we proceed (as in the algorithm in the proof) otherwise (note that we are assuming that the state q_a is the solitary bad state). As regards the proof of **NP**-hardness, we may further assume that every path in any computation tree of M ends in either an accepting node or a rejecting node, where we have designated a particular state of M to be the reject state. The proof then goes through with the only difference being that we associate: the state q_a with a node of T if the node is a rejecting node; and the state q if the node is not a rejecting node (of course, we define $B = \{q_a\}$).

(*c*) and (*d*) With respect to the proof of Theorem 7, we proceed similarly to as in (*a*) and (*b*) above by flipping the association of states to nodes from accepting to rejecting. This suffices to yield the result.

(*e*) Trivial. □

Let us now return to a remark we made when defining the problem MAD. We commented that bounded environments are perhaps not appropriate for maintenance problems in that the problems involve states of affairs which should persist in perpetuity rather than over a 'short' time-span. Of course, if we consider maintenance

problems in unbounded environments then it is not difficult to see that such problems are (generally) undecidable (hint: reduce from the Halting Problem for Turing machines). However, a more sensible approach might be to consider maintenance problems in bounded environments but where the bound is 'longer'.

Define the problem MAD′ as follows: an instance $\mathcal{ENV} = (e_0, E, A, \tau, b)$ is a bounded environment and $B \subseteq E$ is a set of *bad states*; and a yes-instance is an instance for which there exists an agent $\mathcal{AG}$ such that no legitimate run contains a bad state but where the notion of legitimate is with respect to the length of run 2^b (rather than b). Our reasoning is that the exponential bound 2^b, whilst not being 'in perpetuity', might prove to be 'long enough in practice'. The state/action restrictions of the problem MAD′ are defined as expected. Of course, we can also define the problem AAD′ (and its restrictions) in an analogous fashion (but these problems are not so practically motivated).

Define

$$\begin{aligned}
\textbf{NEXPSPACE} &= \cup\{\text{NSPACE}(2^{s(n)}) : s(n) \text{ is some polynomial}\};\\
\textbf{NEXPTIME} &= \cup\{\text{NTIME}(2^{t(n)}) : t(n) \text{ is some polynomial}\};\ \text{and}\\
\textbf{DEXPTIME} &= \cup\{\text{DTIME}(2^{t(n)}) : t(n) \text{ is some polynomial}\}.
\end{aligned}$$

Consider the proofs of the theorems in the previous section and the relevance of the polynomial $t(n)$. This function plays no essential role in the proofs. Indeed, allowing this function to be $2^{q(n)}$, where $q(n)$ is some polynomial, in these proofs, allied with Theorems 1 and 2, immediately yields the following corollary.

Corollary 11 *(a) If $j \geq 3$ and $k \geq 2$ then MAD′(j,k) and AAD′(j,k) are* **NEXPSPACE***-complete.*

(b) If $j = 2$ and $k \geq 2$ then MAD′(j,k) and AAD′(j,k) are **NEXPTIME***-complete.*

(c) If $j \geq 3$ and $k = 1$ then MAD′(j,k) and AAD′(j,k) are **co-NEXPTIME***-complete.*

(d) If $j = 2$ and $k = 1$ then MAD′(j,k) and AAD′(j,k) are **DEXPTIME***-complete.*

(e) If $j = 1$ and $k \geq 1$ then MAD′(j,k) and AAD′(j,k) are trivial. □

We could interpret a bound b as giving rise to runs of length up to 2^{2^b} or $2^{2^{2^b}}$, and so on. Of course, there are analogous results to Corollary 11 in these situations.

6 Conclusion

In this paper we have essentially considered whether imposing numeric criteria on the numbers of states and actions in an agent-based system might make basic achievement and maintenance agent design problems more tractable. We have discovered that even under very severe numeric restrictions these problems remain computationally intractable. Consequently, if we are to stand a chance of making these problems tractable then it must be through restrictions of other sorts. The most obvious restriction to make is on the state transformer function in an environment. In this

paper, we insisted that any state transformer function is always (a description of) a polynomial-time computable function. We could go further and ask if our results still hold when the state transformer function must be logspace computable, or even something yet more restrictive such as first-order definable. Note that, as remarked in our proofs, our results still hold if we insist that any state transformer function must be linear-time computable.

Consider generalizations of our problems where we might consider acceptance criteria more complicated than any run always containing a goal state or no run ever containing a bad state. Essentially, our problems deal with the simplest achievement and maintenance agent design problems we might formulate. Consequently, our lower-bound (completeness) results should apply (in a similar if not an exact fashion) to most other sensible criteria, *e.g.*, such as the criterion considered in [20] where the agent had to aim for a goal state but at the same time avoid every bad state. On the other hand, the **PSPACE** and **NEXPSPACE** upper-bound results (where we do not restrict the numbers of states and actions) will apply to any property definable in any sensible temporal logic, such as CTL, LTL or even quantified CTL (since model checking for these temporal logics can be undertaken in **PSPACE**). We refrain from stating and proving any specific results in this vein as the proofs of any such results would simply be minor extensions of our earlier constructions.

Finally, we remark upon the complexity of optimistic agent-design problems as studied in [20]. In [20], the authors studied a particular achievement agent-design problem where instances are pairs $(\mathcal{ENV}, G)$ (as in the problem AAD) but an agent need only ensure that there is at least one run containing a goal state, as opposed to ensuring that every run should contain a goal state. This problem is known as OPTIMISTIC ACHIEVEMENT AGENT DESIGN (OAD), and it was shown to be **NP**-complete in [20][3] Of course, there are analogously defined problems parameterized by the number of states and the number of actions in an environment. Our proof of Theorem 5 essentially shows that any problem OAD(j, k), for $j \geq 2$ and $k \geq 2$, is **NP**-complete: the environment constructed in that proof is deterministic and so for any agent, the agent ensures that there is at least legitimate run containing a goal state if, and only if, the agent ensures that every legitimate run contains a goal state. Our proof of Theorem 7 essentially shows that any problem OAD$(j, 1)$, for $j \geq 3$, is **NP**-complete: we apply the same construction as in that proof but we work with the usual notion of acceptance for a non-deterministic polynomial-time Turing machine. If we assume (as we did in the proof of Corollary 9) that we work with a deterministic polynomial-time Turing machine in the amended proof of Theorem 7 then we get that the problem OAD$(2, 1)$ is **P**-complete. There are corresponding results for the parameterized versions of the problem OPTIMISTIC MAINTENANCE AGENT DESIGN (OMD).

The situation as regards the problems OAD$'$ and OMD$'$ are similar. The proofs of the above results yield: that OAD$'(j, k)$ and OMD$'(j, k)$ are **NEXPTIME**-complete if $j \geq 2$ and $k \geq 2$ or if $j \geq 3$ and $k = 1$; and that OAD$'(2, 1)$ and OMD$'(2, 1)$ are **DEXPTIME**-complete. Consequently, the results in this paper completely subsume those of [18, 20].

[3]Again, we have an analogous remark to make as we did for the problem AAD earlier regarding bounded environments and runs of length 'polynomial in the size of $|A| \times |E|$'.

References

[1] J.L. Balcázar, J. Díaz and J. Gabarró, *Structural Complexity II*, Springer-Verlag (1990).

[2] C. Baral, V. Kreinovich and R. Trejo, Computational complexity of planning and approximate planning in presence of incompleteness, *Artificial Intelligence* **122** (2000) 241–267.

[3] C. Baral, V. Kreinovich and R. Trejo, Computational complexity of planning with temporal goals, *Proceedings of the* 17*th International Joint Conference on Artificial Intelligence* (ed. B. Nebel) (2001) 509–514.

[4] T. Bylander, The computational complexity of propositional STRIPS planning, *Artificial Intelligence* **69** (1994) 161–204.

[5] M. Cadoli, A survey of complexity results for planning, *Proceedings of the Italian Planning Workshop* 1993 (*IPW'*93) (eds. A. Cesta and S. Gaglio), CNR - National Research Council of Italy - Special Project on Automatic Planning (1993) 131–145.

[6] A.K. Chandra and L.J. Stockmeyer, Alternation, *Proceedings of the* 17*th IEEE Symposium on Foundations of Computer Science* (1976) 98–108.

[7] P.E. Dunne and M. Wooldridge, The computational complexity of agent verification, *Proceedings of Intelligent Agents VIII*: *Agent Theories, Architectures and Languages* (ed. J.-J. Meyer and M. Tambe), Lecture Notes in Artificial Intelligence Volume 2333, Springer-Verlag, Berlin (2002) 115–127.

[8] P.E. Dunne, M.J. Wooldridge and M.R. Laurence, The computational complexity of boolean and stochastic agent design problems, *Proceedings of the* 1*st International Joint Conference on Autonomous Agents and Multiagent Systems*, ACM Press (2002) 976–983.

[9] E.A. Emerson, Temporal and modal Logic, *Handbook of Theoretical Computer Science Volume B* (ed. J. van Leeuwen), Elsevier (1990) 995–1027.

[10] J. Esparza, Decidability and complexity of Petri net problems - an introduction, *Lectures on Petri Nets* I: *Basic Models. Advances in Petri Nets* (eds. G. Rozenberg and W. Reisig), Lecture Notes in Computer Science Volume 1491, Springer-Verlag, Berlin (1998) 374–428.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability*: *A Guide to the Theory of NP-Completeness*, Freeman (1979).

[12] R. Greenlaw, H.J. Hoover, S. Miyano, W.L. Ruzzo, S. Shiraishi and T. Shoudai, *The Parallel Computation Project*: *Volumes* I-III, `http://www.cs.armstrong.edu/greenlaw/parallel.html` (2000).

[13] K. Heljanko, Model checking with finite complete prefixes is PSPACE-complete, *Proceedings of the* 11*th International Conference on Concurrency Theory* (ed. C. Palamidessi), Lecture Notes in Computer Science Volume 1877, Springer-Verlag, Berlin (2000) 108–122.

[14] D. Kozen, On parallelism in Turing machines, *Proceedings of the* 17*th IEEE Symposium on Foundations of Computer Science* (1976) 89–97.

[15] M.L. Littman, J. Goldsmith and M. Mundhenk, The computational complexity of probabilistic planning, *Journal of Artificial Intelligence Research* **9** (1998) 1–36.

[16] P Odifreddi, *Classical Recursion Theory*, North-Holland, Amsterdam (1989).

[17] A.P. Sistla and E.M. Clarke, The complexity of propositional linear temporal logic, *Journal of the Association for Computing Machinery* **32** (1985) 733–749.

[18] M. Wooldridge, The computational complexity of agent design problems, *Proceedings of the Fourth International Conference on Multi-Agent Systems* (ed. E. Durfee), IEEE Press (2000).

[19] M. Wooldridge, On the sources of complexity in agent design, *Applied Artificial Intelligence* **14** (2000) 623–644.

[20] M. Wooldridge and P.E. Dunne, Optimistic and disjunctive agent design problems, *Proceedings of Intelligent Agents VII*: *Agent Theories, Architectures and Languages* (ed. Y. Lesperance and C. Castelfranchi), Lecture Notes in Computer Science Volume 1986, Springer-Verlag, Berlin (2001) 1–14.

[21] M. Wooldridge and N.R. Jennings, Intelligent agents: theory and practice, *The Knowledge Engineering Review* **10** (1995) 115–152.