Reprinted from JOURNAL OF COMPUTER AND SYSTEM SCIENCES
All Rights Reserved by Academic Press, New York and London

Vol. 36, No. 3, June 1988
Printed in Belgium

Probabilistic Game Automata

ANNE CONDON* AND RICHARD E. LADNER[†]

Department of Computer Science, University of Washington, Seattle, Washington 98195

Received September 30, 1987; revised June 25, 1987

We define a *probabilistic game automaton*, a general model of a two-person game. We show how this model includes as special cases the *games against nature* of Papadimitriou [13], the *Arthur–Merlin* games of Babai [1], and the *interactive proof systems* of Goldwasser, Micali, and Rackoff [7]. We prove a number of results about another special case, *games against unknown nature*, which is a generalization of games against nature. In our notation, we let *UP*, (*UC*) denote the class of two-person games with *unbounded* two-sided error where one player plays randomly, with *partial information* (*complete information*). Hence, the designation *UC* refers to games against *known* nature and UP refers to games against *unknown* nature. We show that

$$UC\text{-TIME}(t(n)) \subseteq UP\text{-TIME}(t(n)) \subseteq UC\text{-TIME}(t^2(n)),$$
$$\text{ASPACE}(s(n)) = UC\text{-SPACE}(s(n)) \quad \text{if} \quad s(n) = \Omega(\log n),$$
$$UC\text{-SPACE}(s(n)) \subseteq UP\text{-SPACE}(\log(s(n))) \quad \text{if} \quad s(n) = \Omega(n),$$

where ASPACE($s(n)$) is the class of languages accepted by $s(n)$ space bounded alternating Turing machines. We assume that all the space and time bounds are deterministically constructible. All the inclusions above except one involve the simulation of one game by another. The exception is the result that $UC\text{-SPACE}(s(n)) \subseteq \text{ASPACE}(s(n))$, which is shown by reducing a certain game theoretic problem to linear programming. © 1988 Academic Press, Inc.

1. INTRODUCTION

Because games and game-like phenomena occur naturally in a computational setting, it is natural to formulate many problems in computer science in terms of games. For example, games like chess have been a challenge to researchers in artificial intelligence who desire models of thinking that can be automated. Results on the complexity of logical theories have been proved by using a game-like formulation of logics [3]. More recently, researchers in distributed computing and cryptography have desired models which reflect the competitive nature of distributed and cryptographic protocols.

In order to understand their complexity, various models of computation have

* Supported by an IBM fellowship. Present address: Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53714.

[†] Supported by National Science Foundation Grant DCR-8402565. Some of the research for this paper was done at the Mathematical Sciences Research Institute, Berkeley, California.

been developed which reflect the game-like properties of these problems. These models include the alternating Turing machines of Chandra, Kozen, and Stockmeyer [4], the private alternating Turing machines of Peterson and Reif [14, 15], the games against nature of Papadimitriou [13], the Arthur–Merlin games of Babai [1], and the interactive proof systems of Goldwasser, Micali, and Rackoff [7].

In this paper, we unify and extend the work on these game-like models of computation. We present a new computational model of two-person games, called a *probabilistic game automaton.* The players share an input and have access to states and worktapes. Both players move according to a set of rules, which are modeled as Turing machine-like transition rules. A strategy of a player defines the moves the player makes at any point in the game. There are three important features of games that are included in the definition of probabilistic game automata:

- *Randomness.* Each player can shuffle a deck of cards, roll dice, or flip a coin.
- *Secrecy.* Each player can keep information private from the other player.
- *Power.* Each player can have a limited amount of computational resources in which to carry out its strategy.

Randomness is modeled in a way similar to the way it is modeled in the probabilistic Turing machines of Gill [6], secrecy is modeled in a way similar to the way it is modeled in the private alternating Turing machines of Peterson and Reif [14], and power is modeled by standard time and space bounds and by whether or not nondeterministic choice is allowed. Thus, we can think of a game automaton as a language acceptor where the input is accessible to both the players.

Our goal is to examine the game automaton model carefully to see what the effect is of varying the parameters of randomness, secrecy, and power. It is our hope that its study will give insight into the game-like models already in existence and also shed some light on problems in distributed computing and cryptographic protocols.

Background

The most well-known game-theoretic model of computation is probably the *alternating Turing machine* [4], which models a two-person game of complete information, that is, a game where each player can see all the moves of its opponent. If one player, designated at the outset of the game, has a strategy which always wins, then the input is accepted. The moves of the players model alternation between existential and universal quantifiers. Reif [15] has extended the definition of alternating Turing machines to two-person games of partial information, where a player may not see all the moves of its opponent. Peterson and Reif have considered a restricted form of multi-person games [14]. A special class of games, called *solitaire games*, where one player must play deterministically after its first move, has been studied by Ladner and Norman [12].

Various models of polynomial time bounded games with randomness have also been studied. For example, the *games against nature* of Papadimitriou [13] provide

a natural way to formulate a branch of problems in optimization, i.e., decision problems under uncertainty. In such games, one player, simulating nature, plays randomly and the other player picks a strategy which maximizes its probability of winning against the random player. If this player can win with probability $>\frac{1}{2}$ against the random player, it is considered to have won the game. Recently, Babai [1] defined *Arthur–Merlin games* to be a class of two-person games where Arthur plays randomly. For any input, either Merlin has a strategy which wins with probability $>\frac{3}{4}$ on that input, or every strategy wins with probability at most $\frac{1}{4}$. An input is accepted if Merlin has a strategy which wins with probability $>\frac{3}{4}$ on that input. Babai showed that some computational problems in matrix groups such as membership and order belong to the class of Arthur–Merlin games. The *interactive proof systems* of Goldwasser, Micali, and Rackoff [7] are yet another example of two-person games. These games are similar to Arthur–Merlin games in that one player plays randomly; however, in contrast to Arthur, the random player in an interactive proof system may make moves which cannot be seen by its opponent. Thus interactive proof systems are games of partial information. Related research on interactive proof systems has been on *zero-knowledge proofs.* Such games have applications in cryptography. It should be pointed out that games against nature, Arthur–Merlin games, and interactive proof systems are not two-person games in the traditional game-theoretic sense, because the random player has no control over its moves. However, in keeping with the standard notation in computational complexity theory, these and similar models will be referred to as two-person games in this paper.

The Model

In this paper we introduce a model of a two-person game called a *probabilistic game automaton*, which encompasses these different games into a single uniform framework. The probabilistic game automaton is a model of a two-person game more general than any of the two-person games described above. The two players are named player 0 and player 1. Each player can keep information private from the other player. Some moves taken by either player are *coin-tossing*, where the player flips a coin to determine which next move to make. On other moves, the player can *choose* which move to make from the possible next moves. A move of player 1 made by choice, rather than by flipping a coin, is called an *existential* move. Similarly, a move of player 0 made by choice is called a *universal* move. We use these names to distinguish moves where a player has a choice from the coin-tossing moves. The names are derived from the fact that an existential move models an existential quantifier and a universal move models a universal quantifier. Thus a game consists of a sequence of existential, universal, and coin-tossing moves, in any order.

A *strategy* of a player determines which step a player chooses, based on all the steps of the game automaton so far. To investigate the power of different types of probabilistic game automata we define the automata as language acceptors. The notion of acceptance of an input is defined in terms of strategies for player 1. A strategy for player 1 is a *winning strategy with bound* $\varepsilon \geqslant 0$ if the probability that the strategy leads to a win for player 1, no matter what strategy player 0 uses, is $> \frac{1}{2} + \varepsilon$ and is a *losing strategy with bound* $\varepsilon \geqslant 0$ if the probability that the strategy leads to a win for player 1 is $\leqslant \frac{1}{2} - \varepsilon$. A probabilistic game automaton is *bounded random* if there is an $\varepsilon > 0$ such that for each input there is either a winning strategy for player 1 with bound ε or every strategy is a losing strategy with bound ε. A probabilistic game automaton is *unbounded random* if for each input there is a winning strategy for player 1 with bound 0 or every strategy is a losing strategy with bound 0. The language accepted by a bounded (unbounded) random game automaton is the set of inputs for which player 1 has a winning strategy with bound $\varepsilon > 0$ ($\varepsilon = 0$).

The probabilistic game automaton combines the three features of games mentioned in the introduction—randomness, secrecy, and power—in a natural way. Some features of the model are important in certain applications and not in others. For example, in the study of zero-knowledge proofs, [7], it is essential that both players have private information and can toss coins. In this paper we are simply interested in understanding the complexity of the model, that is, what languages are accepted by the model. Thus we make some simplifying assumptions about the model which make the proofs of this paper simpler to present, without weakening the results.

The first simplifying assumption we make is that player 1 has no private information; that is, all moves of player 1 can be seen by player 0. Because acceptance is defined in terms of strategies for player 1 it turns out that player 1 loses no advantage by making all of its private information visible to player 0. The reason that this is true will become clearer when we describe the model. If player 0 also has no private information then we say that player 0 displays *complete information*. If player 0 has only private information, that is, player 0 reveals nothing to player 1, then we say that player 0 displays *zero information*. In general, player 0 displays *partial information*. The second simplifying assumption we make is that in a game of complete information or partial information, all the coin-tossing moves are made by player 0. Player 0 may be further restricted if it is not allowed to make universal moves but only coin-tossing moves.

We introduce a notation to specify the various types of probabilistic game automata. The symbol $\forall$ is used to denote that player 0 can make universal moves. If M is an unbounded random game automaton, then the letter U is used to denote that player 0 can make random moves; if M is a bounded random game automaton then the letter B is used. Finally the letters Z, P, or C are used to denote that player 0 displays zero, partial, or complete information, respectively. The following table summarizes this notation.

Universal moves	*Random moves*	*Degree of information*
$\forall$	U	Z
	B	P
		C

To specify the restrictions on player 0, at most one symbol is taken from each of the left two columns and exactly one from the third column. For example, *UC* refers to unbounded game automata where player 0 displays complete information and is not allowed universal moves. $\forall C$ refers to game automata where there are no random moves, player 0 can make universal moves and displays complete information; more simply stated, $\forall C$ refers to alternating Turing machines.

Probabilistic game automata can be time bounded or space bounded. For example, UC-TIME$(t(n))$ is the class of languages accepted by UC game automata that run in time bounded by $O(t(n))$. Similarly, UP-SPACE$(s(n))$ is the class of languages accepted by UP game automata that run in space bounded by $O(s(n))$.

New Results

It takes a considerable effort to give a precise definition of probabilistic game automata, but this needs to be done to remove all ambiguity. We show that *UC* game automata are equivalent to the games against nature of Papadimitriou [13], *BC* game automata are equivalent to the Arthur–Merlin games of Babai [1], and the *BP* game automata are equivalent to the interactive proof systems of Goldwasser, Micali, and Rackoff [7].

The main results of this paper concern the unbounded random game automata in the classes *UC* and *UP*. Since *UC* game automata can be thought of as games against nature then we say that *UP* game automata are *games against unknown nature*. The definition of games against unknown nature extends the definition of games against nature, just as the definition of interactive proof systems extends the definition of Arthur–Merlin games. We show that the class of languages accepted by games against unknown nature which run in time $t(n)$, where $t(n)$ is time-constructible, is contained in the class of languages accepted by games against nature which run in time $t^2(n)$. In our notation, for time constructible $t(n) = \Omega(\log n)$,

$$UC\text{-TIME}(t(n)) \subseteq UP\text{-TIME}(t(n)) \subseteq UC\text{-TIME}(t^2(n)).$$

Previously, Papadimitriou [13] showed that the class of languages recognized by alternating Turing machines that run in time $t(n)$ is contained in the class of languages recognized by games against nature that run in time $t(n)$. Yap [16] has recently shown how a language in the class UC-TIME$(t(n))$ can be recognized by an alternating Turing machine running in time $t(n) \log t(n)$. (In fact, Yap's simulation proves the stronger result that $\forall UC$-TIME$(t(n)) \subseteq$ ATIME$(t(n) \log t(n))$). The results of Yap and Papadimitriou show that for time constructible $t(n) = \Omega(n)$,

$$UC\text{-TIME}(t(n)) \subseteq \text{ATIME}(t(n) \log t(n)) \subseteq UC\text{-TIME}(t(n) \log t(n)).$$

From all of these results, it follows that polynomial time bounded alternating Turing machines, games against nature, and games against unknown nature all accept the same class of languages. These results are shown by simulating a time bounded game automaton in one class by a time bounded game automaton in another class.

We also prove new results on space bounded game automata. We show that, unlike time bounded games, if $s(n) = \Omega(n)$ is space construtible then log $s(n)$ space bounded games against unknown nature are powerful enough to simulate $s(n)$ space bounded games against nature. Formally, for space constructible $s(n) = \Omega(n)$,

$$UC\text{-SPACE}(s(n)) \subseteq UP\text{-SPACE}(\log s(n)).$$

We also prove that $s(n)$ space bounded alternating Turing machines accept the same class of languages as $s(n)$ space bounded games against nature. This result is the analog of Papadimitriou's result for time bounded games against nature. Hence for space constructible $s(n) = \Omega(\log n)$,

$$\text{ASPACE}(s(n)) = UC\text{-SPACE}(s(n)).$$

This result is proved using a reduction from a game theoretic problem to linear programming. The other inclusions are shown by simulating one game automaton by another.

In Section 2 we give precise definitions of probabilistic game automata. We relate games such as Arthur–Merlin games, games against nature, and interactive proof systems to the probabilistic game automaton model in Section 3. Sections 4 and 5 contain proofs of our results about unbounded random game automata with time and space bounds, respectively. Finally, conclusions and open problems are presented in Section 6.

2. Definitions

We now describe informally a k-tape probabilistic game automaton, M, with two players, player 0 and player 1. Exactly one player moves during a step of M. There is a special bit called a *turn indicator* which has the value i when the next step of M is made by player i. The states of M are triples from some set $V \times P_0 \times P_1$, where V is a set of visible substates and P_i is a set of substates private to player i, for $i = 0, 1$. Each set V, P_0, or P_1 contains a coin-tossing substate, denoted by vc, p_0c, or p_1c, respectively. A state (v, p_0, p_1) is called a *coin-tossing* state if $v = vc$ or if the value of the turn indicator is i and $p_i = p_ic$. Some subset of the states of M is called the set of *halting* states, which itself is partitioned into *accepting* and *rejecting* states.

The k tapes of M are divided into three disjoint groups: the *visible tapes* and the *tapes private* to player i, for $i = 0, 1$. The input tape is one of the visible tapes. Each player has a *private head* on each of its private tapes and all of the visible tapes. The private head on a visible tape allows the player to read, undetected by the other player, what is written on the visible tape. In addition each player has a *visible head* on each visible tape. M has an input alphabet and a worktape alphabet.

There is a *transition function* δ_i for each player which defines the valid steps of

player i of M. At any moment, player i has exactly two valid steps. A deterministic step is modeled by letting the two steps be the same. The domain of the transition function δ_i is the set of configurations visible to player i. In a step, player i may change the visible substate, player i's private substate, the contents of the tapes under the visible heads and under the heads on the private tapes of player i. Also player i may shift these heads one tape cell to the right or left and may change the turn indicator so that player $1-i$ moves at the next step. Figure 1 shows a probabilistic game automaton with one visible worktape and one private worktape for each player.

We make a distinction between a *step* and a *move*. A step of M is what we have just described, namely an individual step by either player according to its transition function. A move by player i is a sequence of steps which begins just after the turn indicator is first set to i and ends when the turn indicator is set to $1-i$. Hence, a move by a player will consist of a number of steps of that player. In general one player will not know how many steps the other player is taking during the other player's move. This enables a player to do a lot of work privately without the other player knowing anything about how much work has been done.

In order to define a strategy for player 1 and what it means for a probabilistic game automaton to accept an input x, we need the following definitions.

Configurations and Histories

A *configuration* of M on input x is a tuple C, whose components are the current state of M, the turn indicator, the current positions of the heads on the tapes of M and the tape contents. We define visible(C, i) to be a tuple whose components are the part of configuration C which is visible to player i. This consists of the components of C, less the contents of the private tapes of player $1-i$, the private head positions of player $1-i$, and the substate of player $1-i$. Similarly let invisible(C, i) be the part of configuration C which is not visible to player i. Finally, visible(C) is the part of configuration C which is visible to both players. Associated with the transition functions δ_i is a *step relation*, $\rightarrow$, which maps configurations to configurations in the usual way for a Turing machine.

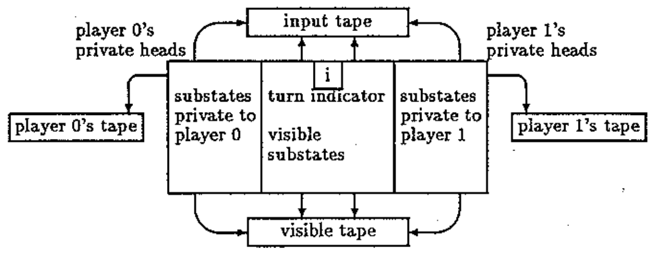

FIG. 1. A probabilistic game automaton with one visible tape and one private tape for each player.

Let player(C) be the value of the turn indicator of configuration C, state(C) be the state of C, and init(M, x) be the initial configuration of M on input x. The configurations are partitioned into a few different types; if state(C) is coin-tossing, accepting, or rejecting we say C is coin-tossing ($\mathscr{R}$), accepting (a), or rejecting (r), respectively. Otherwise if player(C)$=0$ we say C is universal ($\forall$) and if player(C)$=1$ we say C is existential ($\exists$).

A *history* H for M on input x is a sequence $C_0C_1 \cdots C_n$ such that $C_j \to C_{j+1}$ for $0 \leqslant j \leqslant n-1$ and $C_0 = \text{init}(M, x)$. Intuitively, a history describes a sequence of steps of M on x where for $j>0$, component C_j describes the configuration of M after the jth step. We define last(H) to be C_n. Let visible(H, i) be the part of the history visible to player i. Formally, if H is a configuration, visible(H, i) is already defined. Otherwise let HD be a history where H itself is a history. Then

$$\text{visible}(HD, i) = \begin{cases} \text{visible}(H, i), & \text{if visible(last}(H), i) = \text{visible}(D, i), \\ \text{visible}(H, i)\ \text{visible}(D, i), & \text{otherwise.} \end{cases}$$

Finally, let visible(H) be the part of the history H visible to both players. A history H is called a *full history* if state(last(H)) is halting.

Strategies and Computation Trees

Since player 1 does not have access to the private tapes and substates of player 0, player 1's steps can only depend on what player 1 has seen so far in the game. To make this precise we define a *strategy* of player 1 of M on input x to be a function σ mapping histories visible to player 1 into configurations visible to player 1 with the property that if H is a history satisfying player(last(H))$=1$ and state(last(H)) is not a coin-tossing state then HC is a history, where C is a configuration such that σ(visible(H, 1)) = visible(C, 1) and invisible(C, 1) = invisible(last(H), 1). On the other hand, a strategy of player 0 may depend on the history of the game, including the private states and tapes of player 1. The reason for this is that we are interested in seeing how good a strategy of player 1 is against any possible sequence of moves of player 0. We define a strategy of player 0 to be a function τ mapping histories into configurations with the property that if H is a history satisfying player(last(H))$=0$, state(last(H)) is not a coin-tossing state and $\tau(H)=C$ then HC is a history.

For every strategy σ of player 1 of M on x, we define a *computation tree* T_σ to be a (possibly infinite) labeled binary tree with the following properties:

1. Each node η of the tree is labeled with a configuration $l(\eta)$ and the root of the tree has label init(M, x). For a node η, if $l(\eta)$ is a universal, existential, coin-tossing, accepting, or rejecting configuration we say η is a universal, existential, coin-tossing, accepting, or rejecting node, respectively.

2. Any universal or coin-tossing node η has two children θ_1 and θ_2 such that $l(\eta) \to l(\theta_1)$ and $l(\eta) \to l(\theta_2)$.

3. Any existential node η has exactly one child θ and $l(\eta) \to l(\theta)$. Also, if H is a sequence of configurations labeling the nodes of T_σ from the root to η then visible($l(\theta)$, 1) = σ(visible(H, 1)).

4. If η is accepting or rejecting then η is a leaf.

The sequence of configurations labeling nodes of any path from the root of the computation tree is a history. Each computation tree T_σ has a *value* which is a measure of how good strategy σ is. For a tree T_σ define the level, level(η), of a node η to be the distance of that node from the root. The root is at level 0. We refer to the root of T_σ as root(T_σ). For each k we define value(η, k) for each node in the tree as follows: if level(η) $\geqslant k$ then value(η, k) = 0. Otherwise,

$$\text{value}(\eta, k) = \begin{cases} 0, & \text{if } \eta \text{ is rejecting,} \\ 1, & \text{if } \eta \text{ is accepting,} \\ \tfrac{1}{2}[\text{value}(\theta_1, k) + \text{value}(\theta_2, k)], & \text{if } \eta \text{ is cointossing with children } \theta_1, \theta_2, \\ \min[\text{value}(\theta_1, k), \text{value}(\theta_2, k)], & \text{if } \eta \text{ is universal with children } \theta_1, \theta_2, \\ \text{value}(\theta, k), & \text{if } \eta \text{ is existential with child } \theta. \end{cases}$$

It is not difficult to see that for all nodes η and all k:

$$\text{value}(\eta, k) \leqslant \text{value}(\eta, k+1) \leqslant 1.$$

So we define

$$\text{value}(\eta) = \lim_{k \to \infty} \text{value}(\eta, k).$$

The value of computation tree T_σ is denoted by v_σ and is equal to value(root(T_σ)).

It should be clear that if a computation tree has no coin-tossing nodes but consists just of universal and existential nodes, the value of the tree is either 0 or 1. When a computation tree has no universal or existential nodes but just coin-tossing nodes it models a computation where only coin-tossing moves are made, and the value of the tree equals the probability of reaching an accepting leaf of the tree.

Language Acceptance

A strategy σ is a *bounded winning strategy* for player 1 on input x with bound $\varepsilon > 0$, if the value of computation tree T_σ of M on x, denoted by v_σ, is $> \frac{1}{2} + \varepsilon$. Similarly a strategy σ is a *bounded losing strategy* for player 1 on input x with bound $\varepsilon > 0$, if $v_\sigma \leqslant \frac{1}{2} - \varepsilon$. A probabilistic game automaton M is a *bounded random* game automaton if there is $\varepsilon > 0$ (depending only on M) with the property that, for any input x, either player 1 has a bounded winning strategy with bound ε or every strategy of player 1 is a bounded losing strategy with bound ε. Let M be a bounded random game automaton and let $\varepsilon > 0$ be any real number satisfying the definition above. Then the *language accepted by* M is $L(M) = \{x: M$ has a bounded winning strategy for player 1 on input x with bound $\varepsilon\}$.

An *unbounded* winning (or losing) strategy is defined as for a bounded winning (or losing) strategy except $\varepsilon = 0$. A probabilistic game automaton M is an *unbounded random* game automaton if it is a probabilistic game automaton with the property that, for any input x, either player 1 has an unbounded winning strategy or every strategy of player 1 is an unbounded losing strategy. Let M be an unbounded random game automaton. Then the language accepted by M is $L(M) = \{x: M$ has an unbounded winning strategy for player 1 on input $x\}$. Clearly, any language accepted by a bounded random game automaton is also accepted by an unbounded random game automaton.

Time and Space Bounds

In this paper we consider *worst case time and space bounds.* A computation tree is *t time bounded* if the longest path from the root to a leaf is bounded by t. A computation tree is *s space bounded* if each configuration in the tree uses $\leqslant s$ work tape cells. Clearly a time bounded computation tree must be finite, but a space bounded computation tree could possibly be infinite.

A game automaton is *$t(n)$ time bounded* (*$s(n)$ space bounded*) if every strategy for player 1 on each input of length n yields a computation tree which is $t(n)$ time bounded ($s(n)$ space bounded). Because we are only considering time and space bounds that are constructible we could have, without loss of generality, placed our time and space bounds only on winning strategies.

A function $f(n)$ is *time* (*space*) *constructible* if there is a deterministic Turing machine which on each input of length n runs in exactly $t(n)$ time (visits exactly $s(n)$ tape cells) and halts.

Degree of Information

Player 0 may display varying *degrees of information* to player 1. If player 0 never changes its visible substate, never reads or writes on the visible tapes, and never moves its visible heads, we say player 0 displays *zero* information. In such a game, the only action of player 0 which is visible to player 1 is that player 0 changes the turn indicator. If player 0 never changes its private substate and never reads or writes on its private tapes, we say player 0 displays *complete* information. In general player 0 displays *partial* information. A game where player 0 displays complete or zero information is a special case of a game where player 0 displays partial information.

The purpose of this paper is to study what languages are accepted by various types of probabilistic game automata. The notion of acceptance really only depends on the existence (or nonexistence) of good strategies for player 1 against any action of player 0. Since a strategy of player 0 can depend on the complete configuration of the game, including the part of the configuration which is private to player 1, we may as well assume that player 1 has no private information.

When player 0 displays partial or complete information, we can assume that the game automaton has the property that player 1 makes no coin-tossing steps (that is, steps when the turn indicator is 1 and the current state is a coin-tossing state). All the coin-tossing steps for both players can be made by player 0. Henceforth, we will assume that our game automata with complete or partial information are such that player 1 has no private information and does not make coin-tossing steps.

Markov strategies form a special subset of the set of strategies of player 1. In a Markov strategy, player 1's steps depend only on the current state of the game and not on the whole history of the game played so far. Thus we can think of a Markov strategy for player 1 on input x as a function σ mapping configurations visible to player 1 to configurations visible to player 1. More precisely, we say σ is a *Markov* strategy if for any histories H_1 and H_2, if $\text{last}(\text{visible}(H_1, 1)) = \text{last}(\text{visible}(H_2, 1))$, then $\sigma(\text{visible}(H_1, 1)) = \sigma(\text{visible}(H_2, 1))$. It is a standard result from game theory that in a game automaton of complete information, if player 1 has a bounded (unbounded) winning strategy then player 1 has a bounded (unbounded) winning Markov strategy [15]. Intuitively, this is so because at any step of a game of complete information, the complete configuration of the game is visible to player 1, and hence the best move of player 1 can be determined from the configuration. In contrast, at a step of a game of partial information the configuration of the game which is visible to player 1 does not include the private tapes and states of player 0. There may be many possible complete configurations of the game consistent with the configuration visible to player 1. In order for player 1 to determine its next step it must know the probability distribution of the complete configurations at the beginning of the step. The history of the game determines that probability distribution. Thus, player 1's best strategy may depend on the history, not just the current visible configuration.

Notation

Within this general model there are many different types of game automata, where player 1 is existential and player 0 is restricted in some way. To describe the restrictions on player 0, we use the following notation. The symbol $\forall$ is used to denote that player 0 can make universal moves. If M is an unbounded random game automaton, then the letter U is used to denote that player 0 can make coin-tossing moves; if M is a bounded random game automaton then the letter B is used. Finally the letters Z, P, or C are used to denote that player 0 displays zero, partial or complete information, respectively. To specify the restrictions on player 0, $\forall$ is either chosen or not, at most of one of U or B is chosen, and exactly one of Z, P, or C is chosen. For example:

1. A $\forall BP$ automaton is a probabilistic game automaton where player 0 makes universal and coin-tossing moves and displays partial information. On inputs accepted by the automaton, player 1 is required to have a bounded winning strategy, and on inputs not accepted by the automaton, every strategy of player 1 must be a bounded losing strategy.

2. A $\forall Z$ automaton is a game automaton where player 0 can make universal moves but does not make coin-tossing moves. Player 0 displays zero information. On any input accepted by the automaton, player 1 has a strategy which wins against every strategy of player 0.

The suffices –TIME($t(n)$), –SPACE($s(n)$) are used to restrict time, space. For any type of game automata G, G–TIME($t(n)$) is the class of languages accepted by game automata of type G which are $O(t(n))$ time bounded. Similarly, G–SPACE($s(n)$) denotes the class of languages accepted by game automata of type G which are $O(s(n))$ space bounded.

3. Special Cases of Probabilistic Games

Many interesting special cases of probabilistic game automata have already been studied. Some of these can easily be formulated in our general model; for example, the two-person games of incomplete information of Reif [15] and the solitaire games of Ladner and Norman [12]. Another example is Papadimitriou's *games against nature.* A game against nature is a polynomial time game of complete information between two players, one of which is existential and the other of which random, representing nature. There is an input to the game, just as for a probabilistic game automaton, and an input is accepted if the existential player has a strategy which wins against the random player with probability $>\frac{1}{2}$. There is no difficulty extending the definition of games against nature to arbitrary time bounds. Hence a game against nature is a probabilistic game automaton with complete information where player 0 takes no universal steps.

$$\text{GAMES-AGAINST-NATURE}(t(n)) = UC\text{–TIME}(t(n)).$$

Similar to games against nature are the Arthur–Merlin games of Babai [1]. The difference is that the acceptance condition of Arthur–Merlin games on input x requires that the probability of acceptance be bounded away from $\frac{1}{2}$ by some constant $\varepsilon > 0$. If AM–TIME($t(n)$) is the class languages accepted by Arthur–Merlin games running in time $O(t(n))$, then

$$\text{AM–TIME}(t(n)) = BC\text{–TIME}(t(n)).$$

Finally we show how the *interactive proof systems* (IPS) of Goldwasser, Micali, and Rackoff [7] fit into our model. In that paper an IPS is defined in terms of a *pair* of Turing machines; here, for consistency with our other definitions, we define an IPS equivalently as a probabilistic game with partial information. An IPS consists of two players, the *prover*, and the *verifier.* The verifier tosses coins and displays portial information. The players exchange information (called the text of the computation) using the visible tapes. An interactive proof system is denoted by (P, V), where P and V represent the prover and the verifier, respectively. The

verifier of an interactive proof system corresponds to player 0 of a probabilistic game automaton where player 0 takes no universal steps and the prover corresponds to player 1. However, there are three important differences between interactive proof systems and probabilistic game automata:

- the prover of an interactive proof system cannot make existential moves whereas player 1 of a probabilistic game automaton can;
- the time used by the verifier, and *not* the prover, is counted as the time used by an interactive proof system, whereas in a probabilistic game automaton, both the times used by the players 0 and 1 are counted;
- the definition of language acceptance is different for interactive proof systems and probabilistic game automata. Since in an interactive proof system the prover makes no existential moves, it cannot have a strategy. A language L is accepted by interactive proof system (P, V) with bound $\varepsilon > 0$ if

1. for every $x \in L$, (P, V) halts in an accepting state with probability $>\frac{1}{2}+\varepsilon$

and

2. for every $x \notin L$, and any other interactive proof system (P^*, V), (P^*, V) halts in an accepting state with probability $\leqslant \frac{1}{2}-\varepsilon$.

Let IPS–TIME($t(n)$) denote the class of languages accepted by interactive proof systems with time bound $t(n)$. We give a brief argument that

$$\text{IPS–TIME}(t(n)) = BP\text{–TIME}(t(n)).$$

First, suppose L is a language in the class IPS–TIME($t(n)$) and let (P, V) be a $t(n)$ time bounded interactive proof system which accepts L. We define a $t(n)$ time bounded game automaton M in the class BP which accepts L. The verifier V is simulated by player 0 of M and the prover P is simulated by player 1. The problem with this is that player 1 cannot perform all the computations of the prover since this may take time more than $t(n)$. However, note that on a run of the interactive proof, at most $t(n)$ symbols written on the visible tapes by the prover P are read by the verifier V. The idea is that player 1 simulates the prover by existentially writing on the visible tapes the symbols which are read by the verifier. Because it does this existentially, it does not have to do the computations of the prover which would lead to the same symbols being written on the tape. It can do this in $O(t(n))$ time. Since the computation of player 0 is identical to that of the verifier, the probability that M accepts x when player 1's strategy is to simulate prover P equals the probability that (P, V) accepts x. Hence any input accepted by (P, V) is accepted by M. Also if x is not accepted by (P, V) then for all provers P^*, the probability that (P^*, V) halts in an accepting state on x is at most $\frac{1}{2}-\varepsilon$. Thus no matter what strategy player 1 uses, that is, no matter what P^* it simulates, M halts in an accepting state with probability at most $\frac{1}{2}-\varepsilon$ and so x is rejected by M. This shows that M and (P, V) accept the same language.

Conversely suppose L is a language accepted by a $t(n)$ time bounded game automaton M with bound ε in the class BP. We describe an interactive proof (P, V) which accepts L and runs in time $t(n)$. On any input x, player 0 of M is simulated by the verifier V. Player 1 is simulated by the prover. However, the prover cannot make existential moves. Thus to determine which step to take at any time when player 1 would take an existential step, the prover must examine all strategies of player 1 from that step to the end of the game and determine which strategy is best. It can do this by constructing the computation tree for each strategy; the computation tree with greatest value yields the best strategy. It can do this since it has no time limit and the computation trees can be constructed in a straightforward way in time exponential in $t(n)$. The verifier can simulate the steps of player 0, since player 0 uses polynomial time and makes no universal moves. As well as simulating the moves of player 0, the verifier checks after each step of the prover that the prover is properly following player 1's transition function. If the verifier notices that the prover deviates from the transition function of player 1, then the verifier immediately halts in a rejecting state.

If $x \in L$, the prover simulates a strategy of player 1 which has value $> \frac{1}{2} + \varepsilon$ and hence the verifier halts in an accepting state with probability $> \frac{1}{2} + \varepsilon$. However if $x \notin L$, no prover P^* exists for which (P^*, V) accepts x with probability $> \frac{1}{2} - \varepsilon$. To see this, note that the prover P^* can either simulate a strategy of player 1, using the transition function of player 1, or it can deviate from the transition function of player 1. If the prover P^* simulates a strategy of player 1, an accepting state is reached with probability $\leqslant \frac{1}{2} - \varepsilon$ since every strategy of player 1 of M is a bounded losing strategy. The prover P^* cannot increase the probability of reaching an accepting state by deviating from the transition function of player 1, since the verifier checks that P^* is properly following player 1's transition function. Hence x is not accepted by (P^*, V), for any prover P^*.

The verifier V runs in time $O(t(n))$ since player 0 does; hence the time bound of (P, V) is $O(t(n))$. Thus interactive proof systems and bounded random games with the same time bounds are equivalent.

4. The Complexity of Time Bounded Game Automata

There is a close relationship between the complexity of time bounded game automata and alternating Turing machines. Papadimitriou [13], who considered the complexity of unbounded random games, showed that the set of languages accepted by polynomially time bounded games against nature is the same as the set of languages accepted by alternating Turing machines which run in time polynomial in n. Equivalently, UC–TIME(poly(n)) = ATIME(poly(n)), where by poly(n) we mean any polynomial function of n. Yap [16] generalized this result to show that for time constructible $t(n) = \Omega(n)$, $\forall UC$–TIME($t(n)$) $\subseteq$ ATIME($t(n)$ log $t(n)$). The complexity of bounded random game automata with partial information which run in polynomial time, that is, the class BP–TIME(poly(n)) was

studied by Sipser and Goldwasser [8], who showed that BP–TIME(poly(n)) $\subseteq$ BC–TIME(poly(n)).

In this section we consider unbounded random game automata with partial information. Our main result, Theorem 2, proves that the class of languages accepted by unbounded random game automata with *complete* information which run in polynomial time is the same as the class of languages accepted by unbounded random game automata with *partial* information running in polynomial time. The proof technique is different than that used by Sipser and Goldwasser for the bounded random case. We use the following lemma in our proof. In this and all the following proofs we distinguish between the players of distinct automata M and M' by denoting them by player i and player i' respectively for $i = 0, 1$.

Lemma 1. *Let $t(n)$ be time constructible. Any $t(n)$ time bounded probabilistic game automaton M in the class UP can be simulated by a game automaton M' in the class UP which accepts the same language as M, is $O(t^2(n))$ time bounded and has the following properties.*

1. *The players of M' alternate moves at every step.*
2. *All full histories of M' are of the same length.*

Proof. We can assume (see Section 2) that player 1 has no private tapes, heads, or states. The first attempt in describing the simulation of M by M', which is not correct, is as follows. M' simulates M step by step. Suppose at step k of some history of M, player i moves and does not change the turn indicator. Then when M' simulates this step, player i' simulates the step of player i but changes the turn indicator and enters a special visible state. From this state, player $(1-i)'$ may only take a null step, changing the turn indicator and the visible state so that player i' can simulate step $k+1$ of M at the next step of M'.

The problem with this simulation is that there may exist distinct histories H_1, H_2 of M such that visible(H_1) = visible(H_2), but if H'_1 and H'_2 are the histories of M' which simulate H_1, H_2, respectively, then visible(H'_1) $\neq$ visible(H'_2). Hence a strategy of player 1′ may map H'_1 and H'_2 onto distinct configurations, thus increasing the probability that M' halts in an accepting state on input x. To show how such histories H_1 and H_2 can exist, we define a *hidden sequence* of steps of M to be a sequence $i, ..., j$, $i \leqslant j$ of steps of player 0 which has the following properties:

- the visible part of the configurations of M at steps $i, ..., j$ are the same,
- if M does not halt at step j, the visible part of the configuration at step j is different from that of step $j+1$, and
- if $i > 0$, the visible part of the configuration at step $i-1$ is different from that at step i.

Let H_1 and H_2 be histories representing distinct hidden sequences of different lengths such that visible(H_1) = visible(H_2). Then visible(H'_1) = $H'_1 \neq H'_2$ = visible(H'_2), since H'_1 and H'_2 have different lengths.

To overcome this problem, player 0′ must pad all hidden sequences of player 0 to length $t(n)$ during the simulation by taking null steps at which it does not change the visible configuration. Then since player 1′ cannot distinguish the null steps from the simulated steps, all histories of M' which have the same visible part are indistinguishable to player 1′. The padding procedure may square the running time of M' so that it runs in time $O(t^2(n))$.

The automaton M' just constructed satisfies property 1 of the lemma. To ensure that M' satisfies property 2, that is, all histories are of the same length, M' counts the length of the history it is simulating. If M' is about to enter a halting state before $ct^2(n)$ steps, where c is an appropriately chosen constant, M' takes null steps until $ct^2(n)$ steps have passed and then enters the halting state. ▌

THEOREM 2. *If $t(n)$ is time constructible then*

$$UC\text{-TIME}(t(n)) \subseteq UP\text{-TIME}(t(n)) \subseteq UC\text{-TIME}(t^2(n)).$$

Proof. The containment UC-TIME$(t(n)) \subseteq UP$-TIME$(t(n))$ is immediate, since a game automaton with complete information is trivially a game automaton with partial information. Thus we need to show UP-TIME$(t(n)) \subseteq UC$-TIME$(t^2(n))$. The proof is similar to a proof by Reif [15] on games without randomness. From the previous lemma we know that any game automaton in the class UP which is $O(t(n))$ time bounded can be simulated by a game automaton M in the class UP for which every full history has length exactly $ct^2(n)$, for some constant c, and the players alternate turns at every step. Without loss of generality, assume that player 0 of M takes the odd numbered steps and player 1 takes the even numbered steps. We construct a game automaton M' in the class UC which simulates M and is $O(t^2(n))$ time bounded.

Fix an input x and let $m = ct^2(|x|)$. Before we can describe M', we show how a sequence of m numbers, each of constant length, can represent a visible history of M. Given the visible configuration of M on x at time k, there is a constant number, a (assumed to be a power of 2), of possible visible configurations of M at time $k+1$. This is because in changing the visible configuration in one step, a player of M can only change the visible state, the visible tape head positions, and the contents of a constant number of tape cells. The a possible visible configurations can be ordered in a straightforward way so that any number α, $1 \leqslant \alpha \leqslant a$, uniquely determines the αth possible next visible configuration from any given visible configuration.

Let $S = \{\alpha_1 \cdots \alpha_m \mid \alpha_i \in \{1, ..., a\},\ 1 \leqslant i \leqslant m\}$. Each string $\alpha_1 \cdots \alpha_m \in S$ represents a sequence of visible configurations $VC_0 VC_1 \cdots VC_m$, where VC_0 is the initial visible configuration of M and VC_i is the α_ith possible visible configuration from VC_{i-1}. For $1 \leqslant j \leqslant m$ we say $\alpha_1 \cdots \alpha_j$ *represents a visible history* if there is a history $C_0 C_1 \cdots C_j$ of M such that visible$(C_i) = VC_i$, $0 \leqslant i \leqslant j$. A string $\alpha_1 \cdots \alpha_j$ is *valid* if it represents a visible history. The empty string is valid by definition. A string $\alpha_1 \cdots \alpha_m$ is *$\exists$-invalid* if for some even j, $1 \leqslant j \leqslant m$, $\alpha_1 \cdots \alpha_{j-1}$ is valid but $\alpha_1 \cdots \alpha_j$ is not.

Similarly a string $\alpha_1 \cdots \alpha_m$ is *$\mathscr{R}$-invalid* if for some odd j, $1 \leqslant j \leqslant m$, $\alpha_1 \cdots \alpha_{j-1}$ is valid but $\alpha_1 \cdots \alpha_j$ is not. The set S can be partitioned into valid, $\exists$-invalid, and $\mathscr{R}$-invalid strings.

We now describe the simulation of M on x by M'. The simulation is done in two stages. In the first stage, the players of M' write down on a worktape a sequence $\alpha_1 \cdots \alpha_m$ from the set S. If Δ is the worktape alphabet of M, then the worktape alphabet of M' is $\Delta \cup \{1, ..., a\}$. The players write alternate numbers in the sequence. Player 0′ randomly writes down the numbers α_i where i is odd, since these numbers represent configurations reached from cointossing configurations. Similarly player 1′ writes down α_i where i is even. After m turns, a sequence $\alpha_1 \cdots \alpha_m$ is written on the worktape, where for odd i, α_i is written randomly by player 0′, and for even i, α_i is written existentially by player 1′. Each α_i is of constant length, and so the sequence can be written in time $O(m)$. Let $VC_0 \cdots VC_m$ be the visible sequence of configurations represented by $\alpha_1 \cdots \alpha_m$.

The idea of the second stage is that player 0′ tries to simulate a complete history $C_0 C_1 \cdots C_m$ of M, such that visible$(C_i) = VC_i$, $0 \leqslant i \leqslant m$. Clearly this is only possible if $\alpha_1 \cdots \alpha_m$ is valid. Player 1′ does not move in the second stage. Player 0′ starts in the initial configuration of M. Suppose player 0′ has simulated a history $C_0 \cdots C_{j-1}$ such that for $0 \leqslant i \leqslant j-1$, visible$(C_i) = VC_i$. Then player 0′ has simulated the first $j-1$ steps of a history of M and is in configuration C_{j-1}. If j is even, player 0′ checks that $\alpha_1 \cdots \alpha_j$ is valid, given that $\alpha_1 \cdots \alpha_{j-1}$ is. It can do this in constant time. If α_j is not valid then the string $\alpha_1 \cdots \alpha_m$ is $\exists$-invalid and player 0′ halts in a rejecting state. Otherwise player 0′ changes the visible part of configuration C_{j-1} to obtain a new configuration C_j such that visible$(C_j) = VC_j$, the visible configuration represented by α_j.

If j is odd, player 0′ simulates a coin-tossing step of M from configuration C_{j-1}. Thus player 0′ simulates a step of player 0 of M. Let C_j be the configuration of M' after this step. Player 0′ checks if *visible*$(C_j) = VC_j$. If not, player 0′ halts, accepting with probability $\frac{1}{2}$. Otherwise player 0′ continues to the next step of the simulation. If $j = m$ then player 0′ halts, and enters an accepting state if and only if state(C_m) is an accepting state.

This completes the description of M'. Note that the strategy of player 1′ is completely determined in the first stage. Moreover, since player 1′ does not see the part of the history which is private to player 0, its strategy cannot depend on this information.

It is not hard to see that the running time of the automaton M' described above is $O(t^2(n))$ since this is the running time of M. It remains to show that M and M' accept the same language. Fix an input x. The proof that M' accepts x if and only if M does is organized as follows. We first define what it means for a strategy σ' of player 1′ on input x to *simulate* a strategy σ of player 1. A strategy which satisfies this definition is called a simulating strategy. We show that if player 1′ uses a simulating strategy then no string written by the players in the first stage of the simulation is $\exists$-invalid. We use this characterization of a simulating strategy to show that if player 1′ has an unbounded winning strategy, it has one which simulates a strategy of player 1. We then consider the strategies σ' of M' such that σ' simulates some strategy σ of player 1 and show that σ' is an unbounded winning strategy if and only if σ is. Thus M' accepts x if and only if M does.

We consider strategies σ' of player 1′ as mappings from strings $\alpha_1 \cdots \alpha_{j-1}$ where j is even. We say strategy σ' of player 1′ *simulates* strategy σ of player 1 if

$$\sigma'(\alpha_1 \cdots \alpha_{j-1}) = \alpha_j \Leftrightarrow \sigma(VC_0 \cdots VC_{j-1}) = VC_j,$$

for any even j and any valid prefix $\alpha_1 \cdots \alpha_j$ of a string of S which represents visible configurations $VC_0 \cdots VC_j$.

If σ' simulates some strategy σ, we say σ' is a *simulating* strategy. If σ' is a simulating strategy then when a valid string $\alpha_1 \cdots \alpha_{j-1}$ is written in the first stage where j is even, player 1′ writes α_j on the tape where $\alpha_1 \cdots \alpha_j$ is also valid. If $\alpha_1 \cdots \alpha_{j-1}$ is $\mathscr{R}$-invalid, it does not matter what α_j player 1′ writes.

Let $S_{\sigma'}$ be the subset of strings of S of the form $s = \alpha_1 \cdots \alpha_m$ which can be written in the first stage of some execution of M' on x when player 1′ uses strategy σ'.

CLAIM 3. *A strategy σ' is a simulating strategy if and only if $S_{\sigma'}$ has no $\exists$-invalid strings.*

Proof. First suppose σ' simulates strategy σ and suppose $s = \alpha_1 \cdots \alpha_m$ is an $\exists$-invalid string in $S_{\sigma'}$. We show that this leads to a contradiction. Let $VC_0 \cdots VC_m$ be the sequence of visible configurations represented by $\alpha_1 \cdots \alpha_m$. Then for some even j, $VC_0 \cdots VC_{j-1}$ is a visible history of M and $VC_0 \cdots VC_j$ is not. However, since σ' simulates σ, it must be that $\sigma(VC_0 \cdots VC_{j-1}) = VC_j$, contradicting the fact that $VC_0 \cdots VC_j$ is not a visible history of M. To prove the other direction, suppose that $S_{\sigma'}$ has no $\exists$-invalid strings. We wish to show that σ' is a simulating strategy. Let ψ be an arbitrary strategy of player 1 of M. We claim σ' simulates the strategy σ defined as follows on visible history $VC_0 \cdots VC_{j-1}$ where j is even.

$$\sigma(VC_0 \cdots VC_{j-1}) = \begin{cases} VC_j, & \text{if } VC_0 \cdots VC_j \text{ is a visible history} \\ & \text{represented by a prefix of a string in } S_{\sigma'} \\ \psi(VC_0 \cdots VC_{j-1}), & \text{otherwise.} \end{cases}$$

First we show that σ is a well-defined strategy. It is clearly well defined on visible histories which are not represented by a prefix of a string in $S_{\sigma'}$, hence we need only consider the case when $VC_0 \cdots VC_{j-1}$ is represented by a prefix of a string of $S_{\sigma'}$. Suppose s_1 and s_2 are two distinct strings of $S_{\sigma'}$ such that the first $j-1$ numbers of each represent $VC_0 \cdots VC_{j-1}$ where j is even. Then the jth numbers of s_1 and s_2 are the same. This is because the strategy σ' can only depend on the first $j-1$ numbers $\alpha_1, ..., \alpha_{j-1}$ when writing the jth number α_j. Thus VC_j is unique and hence σ' is well defined. It follows immediately from the definition of a simulating strategy that σ' simulates σ. ▌

CLAIM 4. *For any strategy of player 1′ on x, there is always a simulating strategy which is at least as good.*

Proof. Suppose σ'' is not a simulating strategy of player 1′. We define a simulating strategy σ' such that $S_{\sigma'}$ contains all the strings of $S_{\sigma''}$ which are not $\exists$-invalid. To see that such a strategy exists, let ψ' be an arbitrary simulating strategy of player 1′. We define σ' as a function of strings $\alpha_1 \cdots \alpha_{j-1}$, where j is even, as follows:

$$\sigma'(\alpha_1 \cdots \alpha_{j-1}) = \begin{cases} \alpha_j, & \text{if } \alpha_1 \cdots \alpha_j \text{ is not } \exists\text{-invalid} \\ & \text{and is the prefix of a string in } S_{\sigma''} \\ \psi'(\alpha_1 \cdots \alpha_{j-1}), & \text{otherwise.} \end{cases}$$

We need to show that σ' is a simulating strategy and that $v_{\sigma'} \geqslant v_{\sigma''}$. It is straightforward from the definition of σ' to see that $S_{\sigma'}$ has no $\exists$-invalid strings; hence by Claim 3 σ' is a simulating strategy. Next we show that $v_{\sigma'} \geqslant v_{\sigma''}$. For any strategy ψ', if player 1′ uses strategy ψ' then each sequence $s \in S_{\psi'}$ can be written in the first stage of M' with equal probability. This is because all sequences s are of equal length and exactly $\lceil m/2 \rceil$ of the α_i are coin-tossing steps. Hence the probability that M' accepts x when player 1′ uses strategy β' is

$$v_{\beta'} = \frac{1}{|S_{\beta'}|} \sum_{s \in S_{\beta'}} \text{Prob}[M' \text{ accepts } x \text{ if } s \text{ is written in the first stage}].$$

In particular, this formula holds if $\beta' = \sigma''$. The fact that $v_{\sigma''} \leqslant v_{\sigma'}$ follows from the following two observations. First, $|S_{\sigma'}| = |S_{\sigma''}| = a^{\lceil m/2 \rceil}$, since for any given strategy of player 1′, there are $a^{\lceil m/2 \rceil}$ possible strings written in the first stage. Second, from the definition of σ', $S_{\sigma'}$ contains all strings of $S_{\sigma''}$ which are not $\exists$-invalid. Thus

$$\begin{aligned} v_{\sigma''} &= \frac{1}{|S_{\sigma''}|} \sum_{s \in S_{\sigma''}} \text{Prob}[M' \text{ accepts } x \text{ if } s \text{ is written in the first stage}] \\ &= \frac{1}{|S_{\sigma'}|} \sum_{s \in S_{\sigma''}} \text{Prob}[M' \text{ accepts } x \text{ if } s \text{ is written in the first stage}] \\ &\leqslant \frac{1}{|S_{\sigma'}|} \sum_{s \in S_{\sigma'}} \text{Prob}[M' \text{ accepts } x \text{ if } s \text{ is written in the first stage}] \\ &= v_{\sigma'}, \end{aligned}$$

as required. ▌

By Claim 4, we need only consider strategies of M' on x which simulate strategies of M. Let σ' be a strategy of M' which simulates σ. To complete the proof, we derive an expression for the value of $v_{\sigma'}$ in terms of v_{σ}.

CLAIM 5. *If σ' simulates σ then*

$$v_{\sigma'} = \frac{1}{|S_{\sigma'}|}\left(v_\sigma - \frac{1}{2}\right) + \frac{1}{2}.$$

From this it follows immediately that M' accepts x if and only if M does, since $v_{\sigma'} > \frac{1}{2}$ if and only if $v_\sigma > \frac{1}{2}$, and thus σ' is an unbounded winning strategy if and only if σ is. It remains to prove Claim 5.

Proof of Claim 5. Since M satisfies the properties of Lemma 1, the paths in the computation tree T_σ are of equal length and are followed with equal probability. The sequence of labels of each path of T_σ is a history of M. The paths of T_σ can be partitioned into equivalence classes, where two paths are in the same equivalence class if the visible history labeling each of them is equal. Each string s written in the first stage of M' defines a visible history of M. For any string $s \in S_{\sigma'}$, let p_s be the fraction of paths of T_σ which are in the equivalence class corresponding to the visible history represented by s. Let q_s be the fraction of paths in this equivalence class which are accepting. Then the probability of reaching an accepting leaf, following a path from the root of T_σ, is $v_\sigma = \sum_{s \in S_{\sigma'}} p_s q_s$.

Let m be the depth of T_σ. Each path of length m starting at the root of $T_{\sigma'}$ corresponds to a string $s \in S_{\sigma'}$. Each valid path represents a visible history of M. If s is valid or $\mathscr{R}$-invalid we say the corresponding path is valid or $\mathscr{R}$-invalid, respectively. (Since σ' is a simulating strategy, we can assume that $S_{\sigma'}$ has no $\exists$-invalid paths). There is a one-to-one correspondence between the valid paths of $T_{\sigma'}$ and the equivalence classes of paths of T_σ. If a path of $T_{\sigma'}$ corresponds to string s, the subtree T_s rooted at its mth node has one path for each path in the equivalence class corresponding to string s. Altogether, a fraction p_s of the paths of T_s correspond to paths in the equivalence class. Of the other paths of the subtree, the probability of reaching an accepting leaf is $\frac{1}{2}$.

In Fig. 2 there is an example of two computation trees T_σ and $T_{\sigma'}$. The paths of length m from the root of T_σ labeled with "a" are accepting and with "r" are rejecting. Two equivalence classes of paths of T_σ are shown, which correspond to valid paths of $T_{\sigma'}$. The fraction of leaves marked with $*$(respectively $\bullet$) which are accepting is q_{s_1} (respectively q_{s_2}). The path labeled s_3 is $\mathscr{R}$-invalid, hence the probability of reaching an accepting leaf from the root of T_{s_3} is $\frac{1}{2}$.

From this it follows that $p_s q_s + (1 - p_s)\frac{1}{2}$ is the probability of reaching an accepting leaf from the root of the subtree T_s. Thus

$$v_{\sigma'} = \frac{1}{|S_{\sigma'}|} \sum_{s \in S_{\sigma'}} \left(p_s q_s + (1 - p_s)\frac{1}{2}\right) = \frac{1}{|S_{\sigma'}|}\left(v_\sigma - \frac{1}{2}\right) + \frac{1}{2},$$

since $\sum_s p_s = 1$ and $v_\sigma = \sum_s p_s q_s$. This completes the proof of Claim 5. ∎

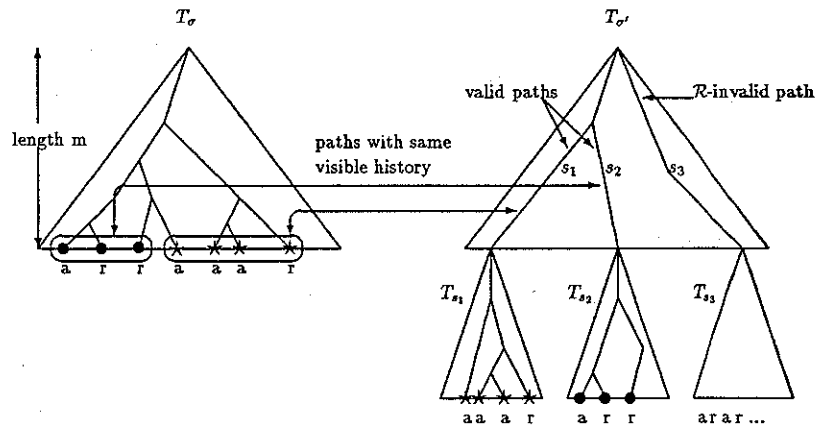

FIG. 2. Computation trees T_σ and $T_{\sigma'}$.

5. THE COMPLEXITY OF SPACE BOUNDED GAME AUTOMATA

The results of this section describe unbounded random space bounded game automata with complete and partial information. We first consider $s(n)$ space bounded games against nature, that is, the class of $s(n)$ space bounded game automata in the class UC. The main result is that for $s(n) = \Omega(\log n)$,

$$UC\text{–SPACE}(s(n)) = \text{ASPACE}(s(n)).$$

This result is the space bounded analog of Papadimitriou's result for time bounded games against nature that UC–TIME(poly(n)) = ATIME(poly(n)). The proof uses a characterization of space bounded game automata in terms of graphs which are examples of *Markov decision processes* (see Howard [9]). We start by describing a mapping from space bounded game automata to graphs. We later show how these graphs relate to general Markov decision processes and prove the main result.

In the final section we consider $s(n)$ space bounded game automata with *partial* information, that is, game automata where player 0 uses private states or tapes. We show that such game automata seem to be more powerful than space bounded game automata with complete information since for $s(n) = \Omega(n)$,

$$UC\text{–SPACE}(s(n)) \subseteq UP\text{–SPACE}(\log s(n)).$$

It is an open problem whether these two classes are equal.

Graph Representation of Game Automata in $\forall UC$–SPACE($s(n)$)

We have seen in Section 2 that any game automaton M on input x can be represented by a tree of all possible computations. If M has space bound $s(n)$ with $s(n)=\Omega(\log n)$, the corresponding tree may have an infinite number of nodes on some inputs, although the number of distinct configurations labeling nodes of the tree is bounded by $d^{s(n)}$, for some constant d. The graph representation of an $s(n)$ space bounded game automaton with complete information which we are about to describe has the advantage of having a finite number, at most $d^{s(n)}$, of nodes.

Let M be an $s(n)$ space bounded game automaton with complete information. Without loss of generality assume that M has a unique accepting and a unique rejecting configuration. We associate with M on input x of length n a directed graph G, having at most $d^{s(n)}$ nodes, for some constant d, where each node is labeled by a distinct configuration of M. Let $\{1, ..., N\}$ be the nodes of G. There is an edge from node i to node j in the graph if there is a transition from the configuration labeling node i to the configuration labeling node j in M. The nodes labeled by accepting or rejecting configurations are called *halting* nodes. Each other node of the graph is either coin-tossing, existential, or universal, depending on the configuration which labels it. All nodes except halting nodes have exactly two outgoing edges; for technical reasons we assume that each halting node i has exactly one reflexive edge (i, i). We call the node labeled by the initial configuration the *start* node.

Consider a subgraph of G obtained by removing one of the two edges from each existential node and each universal node of G. The set of remaining outgoing edges from the existential (universal) nodes of the subgraph is called an *existential policy* σ (*universal policy* τ) of G. We denote the subgraph as $G_{\sigma,\tau}$. There is a one-to-one correspondence between the Markov strategies of player 1 (player 0) of M on input x and the existential (universal) policies of G: hence we use the same symbols to refer to each of them. For each node i of $G_{\sigma,\tau}$, let $v_{\sigma,\tau}(i)$ denote the value of node i, where $v_{\sigma,\tau}(i)$ is the unique value satisfying the following conditions. If there is no path from i to a halting node then $v_{\sigma,\tau}(i)=0$. Otherwise,

$$v_{\sigma,\tau}(i)=\begin{cases} 1, & \text{if } i \text{ is labeled with the accepting configuration,}\\ 0, & \text{if } i \text{ is labeled with the rejecting configuration,}\\ \frac{1}{2}(v_{\sigma,\tau}(j)+v_{\sigma,\tau}(k)), & \text{if } i \text{ is a cointossing node}\\ & \text{with outgoing edges } (i,j), (i,k),\\ v_{\sigma,\tau}(j), & \text{if } i \text{ is an existential or a universal node}\\ & \text{with outgoing edge } (i,j). \end{cases}$$

We show that the values of a graph $G_{\sigma,\tau}$ are well defined. The values of nodes in $G_{\sigma,\tau}$ with no path to a halting node are well defined, as are the values of the halting nodes. Let nodes $1, ..., k$ be the nodes with a path to a halting node and let nodes $N-1$ and N be the halting nodes labeled with the rejecting and accepting configurations, respectively. We use the theory of Markov processes to show that the values of nodes $1 \cdots k$ are well defined. $G_{\sigma,\tau}$ is a Markov process with transition probabilities p_{ij}, $1\leqslant i,j\leqslant N$, defined as follows: $p_{ij}=\frac{1}{2}$ if i is a coin-tossing node with outgoing edge (i,j); $p_{ij}=1$ if i is an existential or universal node with outgoing edge (i,j) and $p_{ij}=0$ otherwise. Also $p_{NN}=p_{N-1N-1}=1$, $p_{Ni}=0$ if $i\neq N$ and $p_{N-1j}=0$ if $j\neq N-1$. Let the $k\times k$ matrix $Q=[p_{ij}]$, $1\leqslant i,j\leqslant k$, be the one-step transition matrix of nodes $1\cdots k$ of the Markov process $G_{\sigma,\tau}$. A property of Q that we will use is that $\lim_{n\to\infty} Q^n=0$. This follows from standard Markov Process theory, based on the fact that the nodes $1, ..., k$ are *transient* since all have a path to a halting node.

Substituting 0 for $v_{\sigma,\tau}(i)$, $k+1\leqslant i\leqslant N-1$, and 1 for $v_{\sigma,\tau}(N)$ in the equations defining $v_{\sigma,\tau}(1),..., v_{\sigma,\tau}(k)$, we have that

$$\mathbf{v}_{\sigma,\tau}=Q\mathbf{v}_{\sigma,\tau}+\mathbf{b},$$

where $\mathbf{b}$ is a constant vector and $\mathbf{v}_{\sigma,\tau}=(v_{\sigma,\tau}(1), ..., v_{\sigma,\tau}(k))^{\mathrm{T}}$. Furthermore, note that each element of $\mathbf{b}$ is a linear combination of $v_{\sigma,\tau}(k+1), ..., v_{\sigma,\tau}(N)$ where all coefficients are nonnegative. Thus $(I-Q)\,\mathbf{v}_{\sigma,\tau}=\mathbf{b}$. There is a unique vector $\mathbf{v}_{\sigma,\tau}$ if and only if $(I-Q)$ has a nonzero determinant. The following proof that $I-Q$ has a nonzero determinant is from Kemeny and Snell [10]. From the basic rules of algebra,

$$(I-Q)(I+Q+\cdots+Q^{n-1})=I-Q^n.$$

Since $\lim_{n\to\infty} Q^n=0$, the limit as $n\to\infty$ of the right-hand side is I which has determinant 1. Hence the determinant of the limit as $n\to\infty$ of the left-hand side is also 1. The determinant of the product of two matrices is the product of the determinants; therefore the determinant of $I-Q$ must be nonzero, as required.

The value $v_{\sigma,\tau}(i)$ of each node of the graph $G_{\sigma,\tau}$ has a natural interpretation in the context of game automaton M. Suppose M is in the configuration which labels node i. Suppose player 1 of M uses the strategy corresponding to existential policy σ and that player 0 of M uses the strategy corresponding to universal policy τ in subsequent moves of the game. Then $v_{\sigma,\tau}(i)$ is the probability that M reaches an accepting state on input x.

Consider the case when M is a space bounded game automaton in the class UC. Then the graph G on input x has no universal nodes and so G has no universal policies. In this case, if σ is an existential policy of G, we denote by G_σ the subgraph G where the edges from the existential nodes are from policy σ and we denote the value of node i of G_σ by $v_\sigma(i)$. Suppose M is in the configuration which labels node i. Suppose player 1 of M uses the strategy corresponding to existential policy σ in subsequent moves of the game. Then $v_\sigma(i)$ is the probability that M reaches an accepting state on input x. The value of T_σ is $v_\sigma=v_\sigma$ (*start* node of G_σ). Since there are a finite number of policies of G, x is accepted by M if and only if $\max_\sigma\{v_\sigma$ (*start* node of G_σ)$\}>\frac{1}{2}$.

Markov Decision Processes

It turns out that the graphs associated with space bounded game automata with complete information can be interpreted as special cases of *Markov decision processes* [9]. A thorough treatment of finite state Markov decision processes is given by Howard in [9]; our definition here is less general than that considered by Howard. A *Markov decision process* $\mathscr{G}$ consists of a set of states $\{1, ..., N\}$, where each state i has a finite set of *choices* E_i. Each choice $p_i \in E_i$ is a vector $(p_{i1}, ..., p_{iN})$, where $\sum_j p_{ij} = 1$. State 1 is called the *start* state, state $N-1$ is the 0-sink state and state N is the 1-sink state. We assume that for all $p_{N-1} \in E_{N-1}, p_{N-1,i} = 0$ for $i \neq N-1$. Similarly, for all $p_N \in E_N$, $p_{N,i} = 0$ for $i \neq N$.

We define a *policy* P of $\mathscr{G}$ to be a matrix $P = [p_{ij}]$ $1 \leqslant i \leqslant N$, where for all i, row i of P is a choice of E_i. The states $\{1, ..., N\}$ together with policy P constitute a Markov process, where p_{ij} is the probability of going from state i to state j in one step.

For a policy P, let values of the states of $\mathscr{G}$, denoted by $\{v_P(i), i = 1, ..., N\}$, be the unique values satisfying the following. If the probability of reaching a sink state from state i is 0, then $v_P(i) = 0$. Otherwise

$$v_P(i) = \begin{cases} 0, & \text{if } i = N-1, \\ 1, & \text{if } i = N, \\ \sum_{j=1}^{N} p_{ij} v_P(j), & \text{otherwise.} \end{cases}$$

CLAIM 6. *The values $v_P(i)$ of $\mathscr{G}$ are well defined and can be evaluated in time polynomial in N.*

Proof. The proof that the values are well defined is exactly like the proof that the values of the subgraph $G_{\sigma,\tau}$ are well defined. It is clearly true for values $v_P(i)$ when the probability of reaching a sink state from state i is 0, since then $v_P(i) = 0$. Also the values of the sink states $N-1$ and N can easily be seen to be well defined. Let states $1, ..., k$ be the states from which the probability of reaching a sink state is greater than zero. Let $\mathbf{v}_P = (v_P(1), ..., v_P(k))^T$. Then just as in Section 5, we can write $\mathbf{v}_P$ as $\mathbf{v}_P = Q\mathbf{v}_P + \mathbf{b}$, where $\mathbf{b}$ is a constant vector and Q is the one-step transition matrix on states $1, ..., k$ of $\mathscr{G}$ with respect to P. The values $v_P(i)$ can be computed in polynomial time by solving the equation $\mathbf{v}_P = Q\mathbf{v}_P + \mathbf{b}$ using any standard method, for example, Cramer's rule [2]. ∎

If M is an $s(n)$ space bounded game automaton in the class UC, the graph G associated with M on input x corresponds to a special kind of Markov decision process. The nodes of G are the states of the Markov decision process. We let $\mathscr{G}$ denote the Markov decision process corresponding to graph G. The start state of $\mathscr{G}$ is the start node of G and the 0- and 1-sink states are the rejecting and accepting nodes, respectively. If i is an existential node of G then state i of $\mathscr{G}$ has two choices, each of the form $p_i = (0, ..., 0, 1, 0, ..., 0)$, where the jth entry of p_i is 1 if (i, j) is an outgoing edge of node i and all other entries of p_i are 0. If i is a cointossing node of G, state i of $\mathscr{G}$ has one choice of the form $p_i = (0, ..., 0, \frac{1}{2}, 0, ..., 0, \frac{1}{2}, 0, ..., 0)$ where the jth and kth entries of p, are $\frac{1}{2}$ if (i, j), (i, k) are the outgoing edges of node i and all other entries of p_i are 0. Each existential policy σ of G corresponds to a policy P of the Markov decision process in a natural way. The definitions of the values of the nodes of graph G with respect to policy σ are consistent with the definitions of the values of the states of the Markov decision process $\mathscr{G}$ with respect to policy P

Space Bounded Game Automata with Complete Information

We can now describe how we will prove the first of our main results on space bounded game automata, that $UC\text{-SPACE}(s(n)) \subseteq \bigcup_{c \geqslant 0} \text{DTIME}(2^{cs(n)})$. Let L be a language in the class UC-SPACE$(s(n))$ and let M be an $s(n)$ space bounded game automaton in the class UC which recognizes L. Let G be the graph representation of M on input x and let $\mathscr{G}$ be the Markov decision process corresponding to graph G. We have already seen at the end of Section 5 that x is accepted by M if and only if

$$\max_{\sigma} \{v_\sigma(\textit{start} \text{ node of } G)\} > \frac{1}{2},$$

where the maximum is taken over all policies σ of graph G. Equivalently, x is accepted by M if and only if

$$\max_{P} \{v_P(\textit{start} \text{ state of } \mathscr{G})\} > \frac{1}{2},$$

where the maximum here is taken over all policies P of $\mathscr{G}$. This is because of the correspondence between policies of graph G and the policies of the Markov decision process $\mathscr{G}$. We say a policy P of $\mathscr{G}$ is *optimal* if v_P (start state of $\mathscr{G}$) $\geqslant v_{P'}$ (*start* state of $\mathscr{G}$) for all policies P' of $\mathscr{G}$. Thus the value of the start state of $\mathscr{G}$, with respect to an optimal policy, is greater than $\frac{1}{2}$ if and only if M accepts x. To prove the theorem, we show how the values of the states of $\mathscr{G}$, with respect to an optimal policy can be computed in time polynomial in the number of states of $\mathscr{G}$. The proof can be broken down into two major steps. First we show that the values $\{v_{\text{opt}}(i), 1 \leqslant i \leqslant N\}$ of the states of $\mathscr{G}$ with respect to some optimal policy are the *minimal* solution to the following equations:

$$v(i) = \begin{cases} \max_{p_i \in E_i} \sum_{j=1}^{N} p_{ij} v(j), & \text{if } 1 \leqslant i \leqslant N-2, \\ 0, & \text{if } i = N-1, \\ 1, & \text{if } i = N. \end{cases} \quad (1)$$

The values $\{v_{\text{opt}}(i)\}$ are a minimal solution to Eqs. (1) in the sense that if $\{v(i)\}$ is any other solution to (1) then $v_{\text{opt}}(i) \leqslant v(i)$, $1 \leqslant i \leqslant N$. The breakdown of this step is as follows. In Lemma 7 we show that the values of the states of $\mathscr{G}$, with respect to *some* policy, satisfy Eqs. (1). Theorem 8 is a technical theorem, from which we derive in Corollary 9 that any policy whose values satisfy Eqs. (1) must be optimal. Corollary 10, which is another corollary of Theorem 8, shows that the values of the states of $\mathscr{G}$, with respect to an optimal policy which satisfy Eqs. (1), must be a *minimal* solution to Eqs. (1), completing the first major step of the proof. The second major step is to show that the minimal solution of Eqs. (1) can be found in time polynomial in N, the number of states of $\mathscr{G}$. This is shown in Theorem 11. Theorem 12 combines the results of all of these lemmas to get the final result.

LEMMA 7. *There is a policy $P^{(\max)}$ of $\mathscr{G}$ such that the values $\{v_{\max}(i), 1 \leqslant i \leqslant N\}$ of the states of G with respect to $P^{(\max)}$* satisfy the Eqs. (1).

Proof. We give an algorithm for constructing $P^{(\max)}$. This algorithm, called the *policy iteration algorithm*, is due to Howard [9]. Unfortunately the algorithm may run in time exponential in the number of states of $\mathscr{G}$. The algorithm proceeds in iterations. There is a *current* policy for each iteration and the current policy for the initial iteration is chosen arbitrarily. At each iteration the algorithm modifies the current policy in a special way to obtain a new policy which becomes the current policy of the next iteration. The algorithm stops when the current policy satisfies Eqs. (1).

Let P' be an arbitrary policy of $\mathscr{G}$.

```
repeat
    $P \leftarrow P'$;
    compute $v_P(i)$ for $1 \leqslant i \leqslant N$;
    if $v_P(i)$ satisfies Eqs. (1) for each $i$ then
        halt and output $P$;
    else
        let $i$ be such that $v_P(i) < \max_{p_i \in E_i} \sum_j p_{ij} v_P(j)$;
        let $p'_i = (p'_{i1}, ..., p'_{iN}) \in E_i$ be such that $\sum p'_{ij} v_P(j) > \sum p_{ij} v_P(j) = v_P(i)$;
        let $P' = [p'_{ij}]$ be such that the $i$th row of $P'$ is $p'_i$ and for $k \neq i$, $p'_{kj} = p_{kj}$.
endrepeat
```

In Claim 6, we showed that the $v_P(i)$ can be computed in time polynomial in N at each iteration. From this it is straightforward to show that each iteration can be completed in time polynomial in N. Clearly if the algorithm halts, it outputs a policy whose values satisfy Eqs. (1). Hence we need only show that the algorithm always halts. To do this we prove the following fact: If P' is the policy obtained from policy P on some iteration of the algorithm then for all k, $v_{P'}(k) \geqslant v_P(k)$ and $\sum_k v_{P'}(k) > \sum_k v_P(k)$.

Let $\Delta_k = v_{P'}(k) - v_P(k)$. Then $\Delta_k = \sum_j p'_{kj} v_{P'}(j) - \sum_j p_{kj} v_P(j)$. Adding and subtracting $\sum p'_{kj} v_P(j)$ we obtain

$$\Delta_k = \sum p'_{kj} v_{P'}(j) - \sum p'_{kj} v_P(j) + \sum p'_{kj} v_P(j) - \sum p_{kj} v_P(j).$$

Let $\delta_k = \sum p'_{kj} v_P(j) - \sum p_{kj} v_P(j)$. Note that $\delta_i > 0$, by the choice of P', and for $k \neq i$, $\delta_k = 0$. Then

$$\Delta_k = \sum_j p'_{kj} \Delta_j + \delta_k.$$

Let $\Delta = (\Delta_1, ..., \Delta_N)^{\mathrm{T}}$ and $\delta = (\delta_1, ..., \Delta_N)^{\mathrm{T}}$. Then $\Delta = P'\Delta + \delta$ which implies that $\Delta = (I - P')^{-1}\delta$. $I - P'$ is invertible since P' is a stochastic matrix; in fact $(I - P')^{-1} = (P')^0 + (P')^1 + \cdots (P')^n ....$ Hence all entries in the vector $(I - P')^{-1}\delta$ are nonnegative. Thus for each k, $\Delta_k = v_{P'}(k) - v_P(k)$ is nonnegative. Moreover, $v_{P'}(i) - v_P(i) > 0$. This is because

$$v_{P'}(i) - v_P(i) = \Delta_i = \sum_j p'_{ij} \Delta_j + \delta_i > 0,$$

since $\delta_i > 0$ and p'_{ij} and $\Delta_j \geqslant 0$ for $1 \leqslant j \leqslant N$. Thus $\sum_k v_{P'}(k) > \sum_k v_P(k)$, completing the proof of the fact.

It is now straightforward to show that the algorithm halts. Since the sum of the values of the current policy at each iteration is strictly greater than that of previous iterations, the current policy is never the same on two different iterations. There are at most 2^N policies so the algorithm must eventually halt and the number of iterations is bounded by the number of policies. Since each iteration takes time polynomial in N, the algorithm runs in worst case time $2^{O(N)}$. ∎

THEOREM 8. *If $\{v_{\text{arb}}(i)\}$ are the values of the states of $\mathscr{G}$ with respect to an arbitrary policy $P^{(\text{arb})}$ then for any nonnegative solution $\{v(i)\}$ of Eqs.* (1), *$v_{\text{arb}}(i) \leqslant v(i)$, for $1 \leqslant i \leqslant N$.*

Proof. By relabeling states if necessary, assume that the probability of reaching a sink state from states $\{k+1, ..., N-2\}$ is 0 and the probability of reaching a sink state from states $\{1, ..., k\}$ is >0 in $\mathscr{G}$ with respect to the policy $P^{(\text{arb})}$. By definition of the values of a policy, $v_{\text{arb}}(k+1) = \cdots = v_{\text{arb}}(N-1) = 0$, $v_{\text{arb}}(N) = 1$ and so $v(i) \geqslant v_{\text{arb}}(i)$ if $i \in \{k+1, ..., N\}$.

It remains to show that $v_{\text{arb}}(i) \leqslant v(i)$, $1 \leqslant i \leqslant k$. Let $\mathbf{v}_{\text{arb}} = (v_{\text{arb}}(1), ..., v_{\text{arb}}(k))^{\mathrm{T}}$. We have already seen that $\mathbf{v}_{\text{arb}} = Q\mathbf{v}_{\text{arb}} + \mathbf{b}$, where Q is the one-step transition matrix on states $1, ..., k$ of $\mathscr{G}$ with respect to $P^{(\text{arb})}$ and $\mathbf{b}$ is a constant vector. Furthermore $(I - Q)$ has nonzero determinant; hence $\mathbf{v}_{\text{arb}} = (I - Q)^{-1}\mathbf{b}$. Each element of $\mathbf{b}$ is a linear combination of $v_{\text{arb}}(k+1),\ldots, v_{\text{arb}}(N)$ with nonnegative coefficients. Let the ith component of the vector $\mathbf{b}$ be $b_i = a_{ik+1}v_{\text{arb}}(k+1) + \cdots + a_{iN}v_{\text{arb}}(N)$.

Similarly if $\mathbf{v} = (v(1),\ldots, v(k))^{\mathrm{T}}$, since the $v(i)$ satisfy Eqs. (1) then $\mathbf{v} \geqslant Q\mathbf{v} + \mathbf{b}'$. Here $\mathbf{b}'$ is a constant vector with component $b'_i = a_{ik+1}v(k+1) + \cdots + a_{iN}v(N)$. Since the coefficients a_{ij} are nonnegative and $v(i) \geqslant v_{\text{arb}}(i)$ for $k+1 \leqslant i \leqslant N$, it follows that $\mathbf{b}' \geqslant \mathbf{b}$. Hence $\mathbf{v} \geqslant Q\mathbf{v} + \mathbf{b}$ and so $\mathbf{v} \geqslant (I-Q)^{-1}\mathbf{b} = \mathbf{v}_{\text{arb}}$. This proves that $v_{\text{arb}}(i) \leqslant v(i)$, $1 \leqslant i \leqslant k$, and we are done. ▌

As an immediate corollary of Theorem 8 we have:

COROLLARY 9. *If the values $\{v_P(i)\}$ of $\mathscr{G}$ with respect to some policy P satisfy Eqs. (1), then P is an optimal policy.*

Proof. From Theorem 8, if $v_{\text{arb}}(i)$ are the values of an arbitrary policy of $\mathscr{G}$, $v_{\text{arb}}(i) \leqslant v_p(i)$ for all i. In particular, for the start state, v_{arb}(*start* state of $\mathscr{G}$) $\leqslant v_P$(*start* state of $\mathscr{G}$); hence P must be an optimal policy. ▌

COROLLARY 10. *If $P^{(\text{opt})}$ is an optimal policy for which the values $\{v_{\text{opt}}(i)\}$ are a solution to Eqs. (1) then $\{v_{\text{opt}}(i)\}$ are minimal nonnegative solutions to Eqs. (1). That is, if $\{v(i)\}$ are any other nonnegative solutions to the equations then $v_{\text{opt}}(i) \leqslant v(i)$, for $1 \leqslant i \leqslant N$.*

Proof. The proof is immediate from Theorem 8. Since the values $\{v_{\text{opt}}(i)\}$ are the values of $\mathscr{G}$ with respect to some policy and the values $\{v(i)\}$ are nonnegative solutions to Eqs. (1), it must be that $v_{\text{opt}}(i) \leqslant v(i)$, $1 \leqslant i \leqslant N$. ▌

THEOREM 11. *The minimal nonnegative solution to Eqs. (1), that is,*

$$v(i) = \begin{cases} \max\limits_{p_i \in E_i} \sum\limits_j p_{ij} v(j), & \text{if } 1 \leqslant i \leqslant N-2, \\ 0, & \text{if } i = N-1, \\ 1, & \text{if } i = N, \end{cases}$$

can be found in time polynomial in N.

Proof. We show that the minimal solution to these equations is the same as the solution to the following linear programming problem: minimize $\sum_{i=1}^{N} v(i)$, subject to the constraints

$$v(i) \geqslant \sum_{j=1}^{N} p_{ij}v(j) \qquad \text{for all } (p_{i1},\ldots,p_{iN}) \in E_i,\ 1 \leqslant i \leqslant N-2,$$

and

$$v(i) \geqslant 0, \qquad 1 \leqslant i \leqslant N-1, \qquad v(N) \geqslant 1.$$

Let $\{v(i),\ 1 \leqslant i \leqslant N\}$ be any solution to the linear programming problem. Note that a solution exists by Lemma 7. Then from the constraints of the linear programming problem it is immediate that

$$v(i) \geqslant \max_{p_i \in E_i} \sum_{j=1}^{N} p_{ij}\, v(j), \qquad v(N-1) \geqslant 0,\ v(N) \geqslant 1.$$

We argue that the $v(i)$ satisfy Eqs. (1) by contradiction. There are three cases to consider: (i) $v(k) > \max_{p_k \in E_k} \sum_{j=1}^{N} p_{kj}v(j)$, for some $k < N-1$; (ii) $v(N-1) > 0$; and (iii) $v(N) > 1$. In each case we construct a vector $\mathbf{v}' = (v'(1),\ldots, v'(N))$ such that $\mathbf{v}'$ satisfies the constraints of the linear programming problem and $\sum v'(i) < \sum v(i)$, contradicting the fact that values $\{v(i),\ 1 \leqslant i \leqslant N\}$ are minimal solutions to the linear programming problem. In the first case let $v'(i) = v(i)$ for $i \neq k$ and let $v'(k) = \max_{p_k \in E_k} \sum_{j=1}^{N} p_{kj}\, v(j)$. In the second case let $v'(N-1) = 0$ and $v'(i) = v(i)$ for $i \neq N-1$. Similarly, for the third case, where $v(N) > 1$, let $v'(N) = 1$, $v'(i) = v(i)$ for $i \neq N$. In all cases $\{v'(i)\}$ satisfies the constraints of the linear programming problem since for $1 \leqslant i \leqslant N-2$, $v'(i) \geqslant \sum_{j=1}^{N} p_{ij}v(j) \geqslant \sum_{j=1}^{N} p_{ij}\, v'(j)$ for all $(p_{i1},\ldots,p_{iN}) \in E_i$ and also $v(N-1) \geqslant 0$, $v(N) \geqslant 1$. Also, since for some i, $v'(i) < v(i)$, $\sum_{i=1}^{N} v'(i) < \sum_{i=1}^{N} v(i)$, which proves the contradiction. Hence an optimal solution to the linear programming problem must satisfy Eqs. (1), and so the minimal solution to Eqs. (1) must be the optimal solution to the linear programming problem. Khachian [11] has shown that the linear programming problem is computable in time polynomial in the length of the input, which is $O(N)$ in this case. Hence the minimal solution to Eqs. (1) can be found in time polynomial in N. ▌

We can finally prove the main theorem of this section.

THEOREM 12. *If $s(n) = \Omega(\log n)$ is constructible then*

$$UC\text{-SPACE}(s(n)) \subseteq \bigcup_{c \geqslant 0} \text{DTIME}(2^{cs(n)}).$$

Proof. Let M be an $s(n)$ space bounded game automaton in the class UC. We describe a $2^{O(s(n))}$ time bounded deterministic Turing machine M' which recognizes the same language as M. On input x, the game automaton M' constructs the Markov decision process $\mathscr{G}$ which corresponds to the graph representing M. This can be done in time polynomial in $d^{s(n)}$, where $d^{s(n)}$ is the number of distinct configurations of M. Let $N = d^{s(n)}$. From Lemma 7 and Corollary 10, the values of the states of $\mathscr{G}$ with respect to some optimal policy satisfy the equations $v(i) = \max_{p_i \in E_i} \sum_{j=1}^{n} p_{ij}v(j)$, $1 \leqslant i \leqslant N-2$, $v(N-1) = 0$, and $v(N) = 1$, and are in fact the minimal solution to these equations. M' finds the minimal solution to these equations in time polynomial in $d^{s(n)}$, using the method of Theorem 11. M' accepts if and only if the value of the *start* state of $\mathscr{G}$ is $> \frac{1}{2}$. Hence the total running time is $2^{O(s(n))}$, as required. ▌

THEOREM. 13. *If $s(n) = \Omega(\log n)$ is space constructible,*

$$\text{ASPACE}(s(n)) \subseteq UC\text{–SPACE}(s(n)).$$

Proof. The proof is similar to the proof of Gill [6] that $NP \subseteq PP$. If M is an $s(n)$ space bounded alternating Turing machine we can assume that the longest history of M is of length $2^{ds(n)}$, for some constant d. On input x of length n, player 0′ of the simulating game automaton M' tosses an unbiased coin with two outcomes, 0 or 1 at the first step. If the outcome is 0, M' simulates M, player 1′ simulating the existential steps of M and player 0′ simulating the universal steps of M, taking coin-tossing steps instead of universal steps. If a 1 is tossed initially then player 0′ tosses $2^{ds(n)}+1$ coins and M' accepts x if and only if the outcome of every coin toss is 1.

If x is accepted by M then x is accepted by M' with probability $\frac{1}{2} + 1/2^{2^{ds(n)}+1} > \frac{1}{2}$. Otherwise the probability that M' accepts x is at most $\frac{1}{2} - 1/2^{2^{ds(n)}} + 1/2^{2^{ds(n)}+1} < \frac{1}{2}$ and so x is not in the language accepted by M'. It is straightforward to see that M' uses space $O(s(n))$. ∎

THEOREM 14. *For $s(n) = \Omega(\log n)$, UC–SPACE$(s(n))$ = ASPACE$(s(n))$.*

Proof. This follows immediately from Theorems 12 and 13. ∎

Space Bounded Game Automata with Partial Information

The techniques used in the above proofs do not extend to space bounded game automata with partial information. Thus no complete characterization of the classes UP–SPACE$(s(n))$ and $\forall UP$–SPACE$(s(n))$ is known. The following theorem shows that space bounded game automata with partial information are likely to be much more powerful than space bounded game automata with complete information.

THEOREM 15. *If $s(n) = \Omega(n)$ is constructible, UC–SPACE$(s(n)) \subseteq$ UP–SPACE$(\log s(n))$.*

Proof. Reif [15] proved a similar result for game automata without randomness; he showed that $\forall C$–SPACE$(s(n)) \subseteq \forall P$–SPACE$(\log s(n))$. The simulation in the proof we present here is similar to Reif's, though the proof is more complicated here. Let M be an $s(n)$ space bounded game automaton in the class UC. In order to prove our result, we assume that M is a one-tape automaton and that the players alternate moves, starting with player 0. We describe a $\log s(n)$ space bounded game automaton M' in the class UP which simulates M.

Intuitively the simulation works as follows. Fix some input x. Player 1′ of M' existentially simulates a history of M on x by listing each symbol of the history in sequence. If player 1′ lists a halting configuration at some step, M' halts in an accepting state if and only if the configuration is accepting. Player 0′ checks that the sequence of symbols listed by player 1′ constitutes a valid history of M and halts in a rejecting state if the history is not valid. Since player 0′ is bounded by $O(\log n)$ space, it cannot check the complete history listed by player 1′. The key idea is that player 0′ *randomly and privately* decides whether to check one symbol of the history. After each step where player 1′ has listed a symbol of the history, player 0′ checks the symbol with probability $\frac{3}{4}$. Once player 0′ has checked a symbol, the computation halts. With probability $\frac{1}{4}$ player 0′ does not check the symbol and player 1′ lists the next symbol of the history. Since player 1′ does not know when player 0′ is checking the listing, it is forced to output a valid history.

We now describe in more detail how player 1′ lists a history of M. A history can be represented as a string $m_0 a_1 m_1 \cdots a_i m_i, \ldots$, where each m_i is a configuration and m_0 is the initial configuration. Each $a_i \in \{1, 2\}$ and $m_{i-1} \rightarrow^{a_i} m_i$ for $i > 0$, that is, the a_ith possible next configuration from m_{i-1} is m_i, according to the transition function of M. Each configuration m_i is represented as a string $c_1 \cdots c_{k-1}\, q\, c_k \cdots c_{s(n)}$, where q is a state of M, $c_1 \cdots c_{s(n)}$ represents the contents of the worktape and the tape head is positioned on the kth tape cell. Each c_i is either an input symbol, a worktape symbol or a special blank symbol. The length of each m_i is $s(n)+1$. The initial configuration m_0 is represented as the string $q_0\, x\, b^{s(n)-|x|}$, where q_0 is the initial state. In this case the string $c_1 \cdots c_n$ represents the input, for $n < i \leqslant s(n)$ c_i is the blank symbol b and $k = 1$.

Let the visible substates of M' contain the worktape and input alphabets of M, the blank symbol and the set $\{1, 2\}$. In one step, player 1′ lists a symbol in the string $m_0 a_1 m_1 \cdots$ by entering the visible substate which corresponds to the symbol. Thus, player 1′ does not use any space at all. In order to list the symbol $a_i \in \{1, 2\}$ when m_{i-1} is a coin-tossing configuration, player 1′ changes the value of the turn indicator and player 0′ takes a cointossing step. Thus a_i is chosen randomly and uniformly if i is odd and is chosen existentially if i is even. As a result, player 1′ lists a_i as a 1 or 2 with equal probability.

Next we show how player 0′ checks that the string listed by player 1′ is a valid history of M. The string listed by player 1′ is *valid* if it satisfies the following conditions:

- m_0 is the initial configuration of M and each m_i has length $s(n)+1$,
- if m_{i-1} is a coin-tossing configuration then a_i is chosen randomly by player 0′ and if m_{i-1} is an existential configuration then a_i is chosen by player 1′, where $a_i \in \{1, 2\}$,
- $m_{i-1} \rightarrow^{a_i} m_i$ for $i > 0$, that is, m_i is the a_ith possible configuration reachable from m_{i-1} according to the transition function.

To check the first condition, player 0′ verifies that m_0 is of the form $q_0 x\, b^{s(n)-|x|}$. To check that a_i is chosen correctly for some i, player 0′ needs to verify that $a_i \in \{1, 2\}$ and that a_i is determined as a result of a coin-tossing step of player 0′ if the turn indicator of configuration m_{i-1} is 0. If player 0′ ever finds that either of the first two conditions is not satisfied, it halts in a rejecting state. To check the last condition, player 0′ would need to write down on a tape configuration m_{i-1} while configuration m_i is being listed by player 1′, in order to check that there is a valid transition from m_{i-1} to m_i. Thus player 0′ cannot check the last condition since each m_i is of length $s(n)+1$ and player 0′ can use space only $O(\log s(n))$. However, player 0′ can check *one* symbol of a configuration as follows. Suppose that player 0′ decides to check the kth symbol of m_i, $i>0$. Then player 0′ stores on a private tape the four symbols numbered $k-1$, k, $k+1$, $k+2$ of configuration m_{i-1}, together with k and a_i. Using this information and the transition function of M, player 0′ can verify that the kth symbol of m_i is valid. Player 0′ uses $O(\log s(n))$ space to store k and constant space to store a_i and the four symbols. The definition of a *valid symbol* follows naturally from the definition of a valid string given by the three conditions above. Player 0′ can check if any symbol of the string listed by player 1′ is valid, using only $O(\log s(n))$ space.

Player 0′ privately and randomly decides to check one symbol of the history listed by player 1′ (excluding the initial configuration) in the following way. Suppose player 1′ lists a symbol, say symbol $k-1$ of configuration m_{i-1} and player 0′ has not already chosen a symbol to check. Then with probability $\frac{3}{4}$ player 0′ decides to check symbol k of configuration m_i and with probability $\frac{1}{4}$ it decides not to check this symbol. In the case that player 0′ decides to check, it records k, the symbol just listed by player 1′ and the next three symbols player 1′ lists. When player 1′ lists a_i player 0′ also records its value. Then at a later time when player 1′ lists the kth symbol of m_i, player 0′ actually checks that the symbol is valid. If the symbol is valid, player 0′ halts in an accepting state with probability $\frac{1}{2}$ and a rejecting state with probability $\frac{1}{2}$. Otherwise the symbol is invalid and player 0′ halts in a rejecting state. If player 0′ does not decide to check the symbol (which happens with probability $\frac{1}{4}$), player 1′ lists the next symbol and player 0′ repeats the same process to decide whether to start checking. It is crucial to the proof that player 1′ does not know whether player 0′ has decided to check a symbol or not.

We summarize this description of M' in the algorithm of Fig. 3. In the algorithm, the boolean variable *checking* is true if and only if player 0′ has decided to check a symbol of some configuration m_i for $i>1$. The variable k records which symbol of the current configuration is being listed by player 1′. The variable *initial* is true only when the first configuration is being listed by player 1′. It is used by player 0′ so that it can check that the initial configuration is correctly listed by player 1′. The variable *oddconfiguration* is true when the configuration listed by player 1′ is an odd numbered configuration. This is used by player 0′ so it can randomly choose a_i for configurations where it is player 0's turn. The variable *checkcount* is used to keep track of which symbols listed by player 1′ need to be recorded by player 0′, and also when the symbol to be checked is actually listed. Since it has value at most $s(n)+1$, it can be implemented in space $O(\log s(n))$. The variable *symbol* denotes the symbol most recently listed by player 1′.

Before getting to the proof that M' accepts the same language as M and that M' is an unbounded random automaton, we introduce some notation. Fix an arbitrary input x. Let σ' be any strategy of player 1′ on x. If the string listed by player 1′ on strategy σ' on *any* sequence of coin tosses of player 0′ is valid, we say σ' is a *valid* strategy. Otherwise σ' is an *invalid* strategy. Each valid strategy σ' of player 1′

```
begin
/* initialization */
oddconfiguration := false; checking := false;
k := 0; initial := true;
repeat
   Player 0′: visibly do the following:
        $k := k + 1 \pmod{s(n) + 2}$;
        if $k = s(n) + 1$ then
            oddconfiguration := not(oddconfiguration);
            if oddconfiguration then with probability $\frac{1}{2}$ $a_i := 1$, else $a_i := 2$;

   Player 1′: existentially list the next symbol of the history being simulated;

   Player 0′: Privately do the following:

        /* check initial configuration */
        if initial then
            if $k = 1$                  then if symbol is not the initial state, halt and reject;
            if $1 < k \le n + 1$        then if symbol is not the $(k - 1)$st input bit, halt and reject;
            if $n + 1 < k \le s(n) + 1$ then if symbol is not the blank symbol, halt and reject;
            if $k = s(n) + 1$           then initial := false;

        if $(k = s(n) + 1)$ and (oddconfiguration) then
            check that symbol equals $a_i$; if not, halt and reject;

        if $(k = s(n) + 1)$ and (not oddconfiguration) then
            check that symbol $\in \{1, 2\}$; if not, halt and reject;

        /* decide whether to start checking */
        if not checking then
            with probability $\frac{3}{4}$ checking := true; checkcount := 0;

        if checking then
            checkcount := checkcount +1;
            if checkcount $\in \{1, \ldots, 4\}$   then record symbol;
            if $k = s(n) + 1$                      then record symbol $a_i$;
            if checkcount $= s(n) + 1$             then check symbol is valid;
                if not valid then halt and reject
                                 else halt, accepting with probability $\frac{1}{2}$
until the last symbol of the history is listed;
if the state of the last configuration listed is accepting then halt and accept
                                                              else halt and reject
end
```

FIG. 3. Algorithm executed by the players of M' in the simulation of M.

corresponds to a strategy of player 1 in a natural way which we now describe. Let H be a history of M ending in an existential configuration and let $m_0 a_1 m_1 \cdots a_i m_i$ represent history H, where i is odd. Then if player 1′ on strategy σ' lists $a_{i+1} m_{i+1}$ after listing $m_0 a_1 \cdots a_i m_i$, define $\sigma(H) = C_{i+1}$ where C_{i+1} is the configuration represented by m_{i+1}. We say σ' *simulates* the strategy σ derived in this way from σ'.

Just as we distinguish between two types of steps of player 1′, we also distinguish between two types of steps of player 0′. At each turn, if player 0′ has not already decided to check a symbol during the current simulation, player 0′ takes either of two actions. With probability $\frac{3}{4}$ it checks that a symbol to be listed later by player 1′ is valid. Alternatively, with probability $\frac{1}{4}$ it does not decide to check. We call a history of M' where player 0′ checks a symbol a *checking history*. A history of M' where player 0′ does not check a symbol is called a *nonchecking history*. Each path of a computation tree $T_{\sigma'}$ is labeled by a history. If the history is a checking history, then the corresponding path in the computation tree is called a checking path; otherwise the path is called a nonchecking path. Each nonchecking path of $T_{\sigma'}$ is a simulation of some history of M; there is a one-to-one correspondence between the nonchecking paths of $T_{\sigma'}$ and the paths of T_σ.

We need to show that player 1′ has a unbounded winning strategy on input x if and only if player 1 does, and that if all strategies of player 1 are unbounded losing strategies, then so also are all strategies of player 1′. The bulk of the proof is divided into the following two claims.

LEMMA 16. *Let M and M' be defined as above and let σ' be a strategy of M' on x which simulates strategy σ of M. Then $v_{\sigma'} > \frac{1}{2}$ if and only if $v_\sigma > \frac{1}{2}$. That is, the probability that M' halts in an accepting state on x when player 1′ uses strategy σ' is $> \frac{1}{2}$ if and only if the probability that M accepts x when player 1 uses strategy σ is $> \frac{1}{2}$.*

LEMMA 17. *Let M and M' be defined as above and let σ' be an invalid strategy of M' on x. Then there is a valid strategy ψ of M' such that $v_{\sigma'} \leqslant v_\psi$.*

Before proving these claims, we show how they can be combined to prove the theorem. Suppose M accepts x. Then some strategy σ of M on x is an unbounded winning strategy, hence $v_\sigma > \frac{1}{2}$. From Claim 16, $v_{\sigma'} > \frac{1}{2}$, where σ' is the strategy of M' which simulates σ. Hence σ' is an unbounded winning strategy and so M' accepts x. The other case to consider is when M rejects x. Then for all strategies σ of M, $v_\sigma \leqslant \frac{1}{2}$. From Claim 16 it follows that all valid strategies σ' of player 1′ must be unbounded losing strategies. Furthermore by Claim 17, all invalid strategies of M' must also be unbounded losing strategies and so x is rejected by M'. Hence M' is an unbounded random automaton and M' accepts the same language as M. We now turn to the proofs of the claims.

Proof of Claim 16. Assume that σ' of M' is a valid strategy and that it simulates strategy σ of M. Note that $v_{\sigma'}$ is the value of the computation tree $T_{\sigma'}$. Recall that we partitioned the paths of $T_{\sigma'}$ into two types: checking paths and nonchecking paths. The probability of reaching an accepting state given that a checking path of $T_{\sigma'}$ is followed is $\frac{1}{2}$. This is because player 0′ halts in an accepting state with probability $\frac{1}{2}$ whenever it checks a symbol. The probability of reaching an accepting state given that a nonchecking path of $T_{\sigma'}$ is followed is v_σ. This is because of the one-to-one correspondence between the nonchecking paths of $T_{\sigma'}$ and the paths of T_σ.

Let p_{check} be the probability that player 0′ checks a symbol listed by player 1′, that is, p_{check} is the probability of following a checking path of $T_{\sigma'}$. Then $v_{\sigma'}$ is

$$v_{\sigma'} = p_{\text{check}} \cdot \frac{1}{2} + (1 - p_{\text{check}}) \cdot v_\sigma.$$

This is greater than $\frac{1}{2}$ if and only if $v_\sigma > \frac{1}{2}$, as required. This completes the proof of Claim 16. ∎

Proof of Claim 17. This claim states that given an invalid strategy σ', there is some valid strategy which is at least as good. Intuitively this is true because there is a stiff penalty for player 1′ when it lists an invalid symbol; if player 0′ checks that symbol, the game automaton halts in a rejecting state. This intuition suggests that by redefining σ' so that player 1′ always lists valid symbols instead of invalid symbols, we get a strategy which has value at least as great as the value of σ'.

We first show how to construct a strategy ψ, which is valid and is the same as σ' on valid histories. Later we argue that the strategy ψ has value at least as great as $v_{\sigma'}$. Without loss of generality we only consider invalid strategies of player 1′ which satisfy the first two conditions of a valid strategy. On any history of the game automaton where player 1′ does not satisfy the first two conditions, the game automaton always halts in a rejecting state, since player 0′ always checks that these conditions are satisfied. Thus let σ' be a strategy of player 1′ which does not satisfy the third condition.

Let ϕ be any valid strategy of M'. Let $\mathscr{VH}$ be the set of valid visible histories VH such that in the transition from VH to $\sigma'(VH)$, player 1′ lists an invalid symbol. We obtain a new strategy ψ by defining ψ to be the same as ϕ on visible histories which have a prefix in $\mathscr{VH}$. We let ψ be the same as σ' on all other histories. Formally,

$$\psi(\text{visible}(H, 1'))$$
$$= \begin{cases} \phi(\text{visible}(H, 1')), & \text{if some } VH \in \mathscr{VH} \text{ is a prefix of visible}(H, 1'), \\ \sigma'(\text{visible}(H, 1')), & \text{otherwise.} \end{cases}$$

It is easy to see that ψ is well defined and is valid. It remains to prove that $v_\psi \geqslant v_{\sigma'}$. We first derive an expression for $v_\psi - v_{\sigma'}$ in terms of the visible histories in $\mathscr{VH}$. For $VH \in \mathscr{VH}$, let prob$[VH]$ be the probability of following a path of $T_{\sigma'}$ which is labeled by a history with visible prefix VH. Then prob$[VH]$ is also the probability of following a path of T_ψ which is labeled by a history with visible prefix VH. This is because the computation trees T_ψ and $T_{\sigma'}$ are identical on paths which are labeled by valid visible histories, and each VH is a valid visible history. Let accept$[\sigma', VH]$ denote the conditional probability of reaching an accepting leaf of $T_{\sigma'}$, given that a path is followed which is labeled by a history with visible prefix VH. Similarly define accept$[\psi, VH]$. We claim that

$$v_\psi - v_{\sigma'} = \sum_{VH \in \mathscr{V}\mathscr{H}} \text{prob}[VH](\text{accept}[\psi, VH] - \text{accept}[\sigma', VH]).$$

To see this, first note that the strategies ψ and σ' differ only on visible histories which have prefix VH for some $VH \in \mathscr{V}\mathscr{H}$. From the definitions of accept$[\sigma', VH]$ and accept$[\psi, VH]$, it folows that accept$[\psi, VH]$ $-$ accept$[\sigma', VH]$ is the difference in the probability that an accepting leaf is reached in computation tree T_ψ and the probability that an accepting leaf is reached in computation tree $T_{\sigma'}$, when following a path of each tree which is labeled by a history with visible prefix VH. Third, the probabilities prob$[VH]$ and prob$[VH']$ are independent for distinct VH, $VH' \in \mathscr{V}\mathscr{H}$. Hence by adding the terms prob$[VH]$(accept$[\psi, VH]$ $-$ accept$[\sigma', VH]$) for all $VH \in \mathscr{V}\mathscr{H}$, the total difference between v_ψ and $v_{\sigma'}$ is obtained.

We now show that accept$[\psi, VH]$ $-$ accept$[\sigma', VH] > 0$ for any visible history $VH \in \mathscr{V}\mathscr{H}$. Fix some $VH \in \mathscr{V}\mathscr{H}$. Suppose the symbol listed by player 1′ in the transition from VH to $\sigma'(VH)$ is the kth symbol of m_i for some k and i. If VH has occurred, player 0′ cannot have decided to check any symbol listed *before* the kth symbol of m_i. This is because as soon as player 0′ checks a symbol, it halts, and player 1′ lists no further symbols. Since player 0′ has not already decided to check a symbol, the probability that player 0′ checks the kth symbol of m_i is $\frac{3}{4}$. If player 0′ checks this symbol, it halts in a rejecting state when player 1′ uses strategy σ' because player 1′ lists an invalid symbol in the transition from VH to $\sigma'(VH)$. This proves that accept$[\sigma', VH] \leqslant \frac{1}{4}$.

In a similar way, we prove that $\frac{1}{4} <$ accept$[\psi, VH]$. If player 1′ uses strategy ψ, it lists a valid symbol in the transition from VH to $\psi(VH)$. With probability $\frac{3}{4}$ player 0′ checks that the symbol is valid and if it is, it halts in an accepting state with probability $\frac{1}{2}$. This means that the probability M halts in an accepting state is $\geqslant \frac{3}{4} \cdot \frac{1}{2} > \frac{1}{4}$ and so $\frac{1}{4} <$ accept$[\psi, VH]$. We have now shown that for any $VH \in \mathscr{V}\mathscr{H}$, accept$[\sigma', VH] \leqslant \frac{1}{4} <$ accept$[\psi, VH]$. Since prob[VH] $\geqslant 0$ for all $VH \in \mathscr{V}\mathscr{H}$, it follows that

$$v_\psi - v_{\sigma'} = \sum_{VH \in \mathscr{V}\mathscr{H}} \text{prob}[VH](\text{accept}[\psi, VH] - \text{accept}[\sigma', VH]) \geqslant 0.$$

This completes the proof that $v_\psi \geqslant v_{\sigma'}$, and so the claim is proved. ∎

6. Conclusion

The probabilistic game automaton provides a uniform framework for the study of game-like phenomena in a computational setting. We have given a precise description of the probabilistic game automaton, and have shown how it includes as special cases Arthur–Merlin games [1], interactive proof systems [7], and other game classes studied in the computer science literature [12, 13, 15]. We have proved results for special classes of games, mainly the unbounded random game automata which model games against nature and games against unknown nature. In particular, we have shown that the class of languages accepted by polynomial time bounded games against nature is the same as the class of languages accepted by polynomial time bounded games against unknown nature. However, the class of languages accepted polynomial space bounded games against nature is contained in the class of languages accepted by logarithmic space bounded games against unknown nature.

Some new results on bounded random game automata have been proved and appear in [5]. In that paper, the class of languages accepted by space bounded Arthur–Merlin games for space constructible $s(n)$, that is, the class BC–SPACE$(s(n))$, is shown to be equal to ASPACE$(s(n))$. This result, together with Theorem 14 of this paper, implies that Arthur–Merlin games and games against nature with the same space bounds are equivalent. Recall that the difference between these models is that Arthur–Merlin games have error probability bounded away from $\frac{1}{2}$ whereas games against nature do not. This is the first example known to us of a probabilistic complexity class which is invariant under the definition of error probability. Theorem 15 of this paper is extended in [5] to show that interactive proof systems with space bound log $s(n)$ can simulate Arthur–Merlin games with space bound $s(n)$, where $s(n) = \Omega(n)$ is any space-constructible function. Thus

$$BC\text{–SPACE}(s(n)) \subseteq BP\text{–SPACE}(\log s(n)).$$

There still remain many open problems on the complexity of game automata. First, we would like to find an improvement in our simulation of Theorem 2, where we eliminate partial information from unbounded random game automata. Is there a way to simulate an unbounded random game automaton with partial information by one with complete information, without squaring the running time?

Another problem which has not been resolved is whether

$$UC\text{–SPACE}(s(n)) = UP\text{–SPACE}(\log s(n)).$$

In Theorem 15 we showed that UC–SPACE$(s(n)) \subseteq UP$–SPACE$(\log s(n))$ but we have not proved the other direction. We would also like to extend the result of Theorem 10 to show that $\forall UC$–SPACE$(s(n)) \subseteq$ ASPACE$(s(n))$. We can show that $\forall UC$–SPACE$(s(n)) \subseteq \bigcup_{c \geqslant 0}$ NTIME$(2^{cs(n)})$, but we conjecture that these classes are not equal. An interesting question posed by Babai [1] is whether BC–TIME(poly$(n)) \subseteq \Sigma_k^P$, where poly$(n)$ is any polynomial function of n and Σ_k^P is the class of polynomial time bounded alternating Turing machines with k alternations, starting with the existential player. Finally we have not looked at game automata with partial information, where player 0 makes both random and universal moves. Can the results of this paper be extended to such game automata?

ACKNOWLEDGMENTS

We would like to thank Richard Anderson and Nimrod Megiddo for some useful discussions concerning the proof of Theorem 12. We would also like to thank the anonymous referees for their careful reading of this paper; their comments have greatly improved the presentation.

REFERENCES

1. L. BABAI, Trading group theory for randomness, *in* "Proceedings, 17th ACM Symp. Theory of Computing, May 1985," pp. 421–429.
2. R. W. BALL AND R. A. BEAUMONT, "Introduction to Modern Algebra and Matrix Theory," Reinhart, New York, 1956.
3. L. BERMAN, The complexity of logical theories, *Theoret. Comput. Sci.* **11** (1980), 71–77.
4. A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, Alternation, *J. Assoc. Comput. Mach.* No. **1** (1981), 114–133.
5. A. CONDON, "Space Bounded Probabilistic Games," Technical report, Number 87-01-04, University of Washington, Seattle, 1987.
6. J. GILL, The computational complexity of probabilistic Turing machines, *SIAM J. Comput.* **6** (1977), 675–695.
7. S. GOLDWASSER, S. MICALI, AND C. RACKOFF, The knowledge complexity of interactive protocols, *in* "Proceedings, 17th ACM Symp. Theory of Computing, May 1985," pp. 291–304.
8. S. GOLDWASSER AND M. SIPSER, Private coins versus public coins in interactive proof systems, *in* "Proceedings, 18th ACM Symp. Theory of Computing, May 1986," pp. 59–68.
9. HOWARD, "Dynamic Programming and Markov Processes," MIT Press, Cambridge, MA, 1980.
10. J. G. KEMENY AND J. L. SNELL, "Finite Markov Chains," Van Nostrand, Princeton, NJ, 1960.
11. L. G. KHACHIYAN, A polynomial algorithm in linear programming, *Soviet Math Dokl.* **20** (1979), 191–194.
12. R. E. LADNER AND J. K. NORMAN, Solitaire automata, *J. Comput. System Sci.* **30**, No. 1 (1985), 116–129.
13. C. H. PAPADIMITRIOU, Games against nature, *in* "Proceedings, 24th IEEE Symp. Found. of Comput. Sci., 1983," pp. 446–450.
14. G. L. PETERSON AND J. H. REIF, Multiple person alternation, *in* "Proceedings, 20th IEEE Symp. Found. of Comput. Sci., 1979," pp. 348–363.
15. J. H. REIF, The complexity of two-player games of incomplete information, *J. Comput. System Sci.* **29**, No. 2 (1984), 274–301.
16. C. YAP, Valuation machines, manuscript from New York University.

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium