

Memory-Bounded Game-Playing Computing Devices

R.E. Stearns

Department of Computer Science

University at Albany

Albany, N.Y. 12222

July 3, 1989

1 Introduction

There has been a recent interest in machines which play two-person repeated games, especially machines which have bounded memory [Au], [KS], [Ru]. Any machine with a bounded memory can only have a finite set of states and can, in principle, be modeled with a finite state automaton. The number of states is an obvious measure of a strategy's complexity, and some interesting results using this measure have been obtained [B-P], [Ne]. In particular, it has been shown that a strategy which "defeats" all n -state automata requires a number of states which is exponential in n .

In this paper, we consider measuring the complexity of a strategy based on the number of bits required to implement the strategy on some fixed general purpose computer. Our main result is that there exists a strategy which "defeats" all b -bit strategies on a given computer and which uses a number of bits which is linear in b . More specifically, the number of bits used is a linear function $Ab+B$ where constant A depends on the percentage of the time the winning player is guaranteed to make his

best response and constant B depends only on the computer which runs the defeated strategy.

Definition 1.1. A **game** G consists of two finite **move** sets M_I and M_{II} for players I and II and a **payoff** function $\pi : M_I \times M_{II} \rightarrow REALS$ for player I. Define r to be a mapping $r : M_{II} \rightarrow M_I$ such that $r(m)$ maximizes $\pi(r(m), m)$. ■

Definition 1.2. A **player I strategy** σ_I for repeated G is a rule which, for all $p \geq 1$, puts out a move m_p from M_I , receives as input a move m'_p from M_{II} , and computes a next move m_{p+1} from M_I . A **player II strategy** σ_{II} is defined similarly. Strategies σ_I and σ_{II} together produces an infinite sequence $(m_1, m'_1), (m_2, m'_2), \dots$ called the **result** of σ_I and σ_{II} . ■

In the case that σ_I and σ_{II} are played by machines A_I and A_{II} having only a finite number of states, A_I and A_{II} connected together have only a finite number of states, and the sequence they produce is periodic. The resulting sequence of payoffs $\pi(m_p, m'_p)$ thus has a well defined value under any of the customary methods of computing average values for infinite sequences. If the machine A_I is successful in playing $r(m'_p)$ a high percentage of the time, then player I is guaranteed a payoff close to $v = MIN_{m' \in S_{II}} MAX_{m \in S_I} \pi(m, m')$ (i.e. the min max over pure strategies). Player II can insure player I gets no more than v using the one-state strategy of always playing a m' which minimizes $\pi(r(m'), m')$.

Given a game G , we are interested in the super games where each player selects a machine to play his strategy σ , the machines then play G repeatedly, and player II pays player I the average payoff from the repeated play. When player II is restricted in his choice of machines because of memory limits, we are interested in machines for player I which will “defeat” player II regardless of the machine player II chooses. By “defeat”, we mean guarantee that player I gets close to v . This will be obtained by playing $m_p = r(m'_p)$ a high percentage of the time. A stronger concept of “defeat” is to play $m_p = r(m'_p)$ all but a finite number of times. This guarantees at least v . An even stronger concept is possible based on generating the most favorable periodic sequence of payoffs that the opponent’s machine will permit.

The concepts under consideration can be illustrated with the game of “matching pennies”. In this game $M_I = M_{II} = \{H, T\}$, $\pi(H, H) = \pi(T, T) = 1$, $\pi(H, T) = \pi(T, H) = -1$, $r(H) = H$ and $r(T) = T$. In the repeated game, if player I matches the choices of player II most of the time, then the average payoff for player I will be close to one. If the players had access to a random device during the play, the players could of course insure an average outcome of zero by playing H and T each with a probability of one-half. But the machines which play the game behave deterministically and are thus incapable of truly random behavior. The players can introduce an element of true randomness in the beginning by selecting their machines randomly, but this choice may be discovered during the course of play and exploited by the opponent’s machine.

To appreciate some issues concerning memory measurement, consider three finite state strategies for player I playing matching pennies:

Strategy A. Play $m_p = H$ if $P \equiv 0 \pmod{10^6}$ and $m_p = T$ otherwise.

Strategy B. Play $m_p = H$ for $p \leq 10^6$ and $m_p = m'_{p-10^6}$ otherwise.

Strategy C. Use a table of 10^6 “random” bits and choose m_p according to bit i where $i \equiv p \pmod{10^6}$.

Strategies A and C each have a million states whereas strategy B has $2^{1000000}$ states. Thus by counting states, A and C seem about equal whereas B seems much more complex.

The complexities seem much different if we consider programming these strategies on a computer. Strategy A can be implemented with a very short program and a tiny amount of additional memory (20 bits) to remember $p \pmod{10^6}$. Strategy B can also be implemented with a very short program but requires an additional 10^6 bits of memory to remember the last 10^6 moves of the other player. The program for Strategy C requires 10^6 bits since the table of 10^6 bits must be included as part of the program. But the additional memory required by C for program execution is very small and is similar to strategy A.

Comparing the resources needed to execute strategies B and C, each requires a

little more than 10^6 bits. In the case of Strategy C, most of the bits are set when the program is loaded and remain unchanged during the play of the super game. Because they do not change, these bits do not contribute to the state count. In the case of Strategy B, the bits can change during play and thus do affect the state count. The approach we take in this paper will treat a bit as a resource regardless of whether the bit changes during the play of the game. Thus, in contrast to the conclusions suggested by state counting, we consider Strategies B and C to have about the same complexity and Strategy A to be much simpler.

The model we use is based on the above discussion. The formal model will be introduced in the next section. Players I and II will each have a computing device which can, in principle, operate on any size memory. Think of each player's device as a computer which will be programmed to play a strategy. To play the game, each player provides his computer with a fixed amount of memory and initializes the memory in any way he sees fit that results in a well-defined strategy. By "well-defined", we mean that the computer comes up with a next move for any finite sequence of moves by the opponent. This could fail to happen if the computer got caught in a loop sometime without producing a new output. After initialization, the computers play the repeated game. Each machine will be playing a finite-state strategy because of the memory limitation.

We take the number of memory bits provided as the complexity of the program which implements the strategy. This is an upper bound on the complexity of the strategy itself which we define as the least amount of memory required to implement the strategy. Obviously, this measure depends on the particular computer on which the strategy is implemented. This allows us to talk about $B_{\mathcal{A}}(\sigma)$, the number of bits required to implement a finite-state strategy σ on a computer \mathcal{A} .

Although we are placing strong emphasis on providing memory and initializing memory, we still have the underlying computers to consider. Specifically, we need to understand the strategic implications of the players choosing computers on which to program their strategies. It turns out that this is a relatively minor consideration.

If a player has a sufficiently powerful computer, it makes very little difference what computer his opponent has. Even if player I picks his computer first and then player II picks his computer, knowing the choice of player I, the amount of memory player I needs to defeat player II is linear in the memory used by player II. The complexity due to player II's computer is an additive constant representing the number of bits needed to describe player II's computer.

In regard to measuring the complexity of a strategy, the bound obtained with respect to a “sufficiently powerful computer” is also within an additive constant of the bound achieved by some other computer. Symbolically, there is a computer \mathcal{U} such that, for all computers \mathcal{A} , there is a constant $K_{\mathcal{A}}$ such that

$$B_{\mathcal{U}}(\sigma) \leq B_{\mathcal{A}}(\sigma) + K_{\mathcal{A}}$$

for all strategies σ . Thus counting memory bits is a reasonable framework for discussing complexity, even though there are many plausible choices for \mathcal{U} and consequently no exact answer to how many bits are required for a particular strategy.

The approach we are taking to the complexity of strategies follows the approach of Kolmogorov in measuring the complexity or randomness of a finite sequence [Ko], [Le], and [So]. The properties of our measure and the proofs of these properties are easily anticipated by those familiar with this approach.

2 The Computer Model

The model of a memory bounded game-playing machine that we use is based on a Turing machine or (more specifically) a linear-bounded automaton (LBA). In computer science, the term “Turing machine” is used generically to refer to any kind of a device where a finite state machine is connected to a tape (or tapes). The tape is made up of “squares”, one for each integer. Each square contains a symbol from a finite tape alphabet. The finite state machine is regarded as having a “head” which reads one tape square at a time. The machine makes a “transition” according to the

tape symbol located at the position of the head and the current state of the finite-state machine. The transition consists of changing state, changing the tape symbol, and (optionally) moving the head one square to the left or right.

In Turing's original work, his machines were started on a blank tape on which an integer had been written. This integer was regarded as input to the machine. Certain states were designated as halting states. The machine was started in a specified "starting state" with the head on the left-most symbol of the input integer. If, after a series of transitions, the machine reached a halting state, the contents of the tape was interpreted as an integer value computed by the machine. This integer was regarded as output of the machine. Subsequent to Turing's work, many different ways of defining inputs and outputs have been employed, depending on the application, but the name "Turing machine" has been retained. In our case, we will adjust the definition so that the machine receives information about the opponent's moves and puts out its own moves.

When a Turing machine is restricted so that only a bounded amount of tape is provided, it is generally called a "linear-bounded automaton" or LBA for short. The input is presented to the LBA as a sequence of symbols written between a "left endmarker" and a "right endmarker". The transitions are restricted so as to never move the head left of a left endmarker or right of a right endmarker. Our computer model can be described as an LBA adapted to playing games. When we wish to count bits, we will restrict the symbols between the endmarkers to be from the set $\{0,1\}$.

We now present our formal model. This is necessary so as to have formal theorems. However, the results do not depend in any significant way on the details of this model. These details are somewhat arbitrary and many variations on this model would give the same results:

Definition 2.1. A **game-playing LBA** is specified by a six-tuple $(M_1, M_2, S, \alpha, \Gamma, \tau)$ where

- 1) M_1 and M_2 are finite sets of **input moves** and **output moves**

- 2) S is a finite set of **states**
- 3) α is an **action** mapping $\alpha : S \rightarrow M_1 \cup \{CONTINUE\}$
- 4) Γ is a finite set of tape symbols which includes $\{\$, \phi\}$ (the **endmarkers**)
- 5) $\tau : M_2 \times S \times \Gamma \rightarrow S \times \Gamma \times \{\mathbf{L}, \mathbf{R}, \mathbf{S}\}$ is the **transition function** and is subject to the condition that, whenever $\tau(m, s, t) \rightarrow (s', t', \mathbf{D})$, $t = \$$ iff $t' = \$$, $t = \phi$ iff $t' = \phi$, $t = \$$ implies $\mathbf{D} \neq \mathbf{L}$, and $t = \phi$ implies $\mathbf{D} \neq \mathbf{R}$.

If $\Gamma = \{0, 1, \$, \phi\}$ we say the LBA is **binary**. ■

The action function α specifies whether the machine in state s is ready to make a move $\alpha(s)$ or is required to continue computing in order to select its next move. The transition specifies how the machine is to behave when it is responding to move m and is in state s with tape symbol t under its head. The LBA replaces t with t' on the tape, moves its head depending on the selection from $\{\mathbf{L}, \mathbf{R}, \mathbf{S}\}$, and changes to state s' . \mathbf{L} means “move **left** one square”, \mathbf{R} means “move **right** one square”, \mathbf{S} means “**stay** on the same square”. The conditions on the transition function insure that the head cannot be moved left of $\$$ or right of ϕ , and that $\$$ and ϕ continue to appear only at their respective ends of the tape.

During the play of a super game, a game-playing LBA will go through a sequence of “configurations”. By “configuration”, we mean a combination of state, tape contents, head position, and input move. For the sake of formal proofs, we need a formal definition of this concept. Many ways are possible and we choose the following:

Definition 2.2. Given a game-playing LBA $\mathcal{A} = (M_1, M_2, S, \alpha, \Gamma, \tau)$, we say that (m, s, w, t, x) is a configuration of \mathcal{A} if and only if

- 1) m is in M_2
- 2) s is in S
- 3) w and x are in Γ^* (Γ^* is the set of finite strings on Γ)
- 4) t is in Γ

5) the string $w\alpha x$ begins with $\$$, ends with ϕ , and these symbols do not appear elsewhere in $w\alpha x$.

The configuration is called an **intermediate configuration** if $\alpha(s) = CONTINUE$ and is called an **output configuration** if $\alpha(s)$ is in M_1 . ■

In an output configuration, $\alpha(s)$ is interpreted as the move selected by the LBA and the m in M_2 is interpreted as the move selected by the other player. If the LBA is playing for player I, then $M_1 = M_I$, $M_2 = M_{II}$, and pair $(\alpha(s), m)$ is the next pair in the result sequence. If the LBA is playing for player II, then $M_1 = M_{II}$, $M_2 = M_I$, and pair $(m, \alpha(s))$ is the next pair in the result sequence.

In an intermediate configuration, the LBA is considered to be in the process of computing its next game move. Since m represents the opponents last move and since moves are to be made simultaneously after both players have selected their moves, the m in any intermediate configuration represents the same move as in the previous configuration, so m must be the same as in the previous configuration.

A player equipped with an LBA begins play by picking some initial sequence for his tape, selecting an initial head position, and selecting a state s such that $\alpha(s)$ is in M_1 . The state, tape and head position, together with the first move of the opponent, make up the initial configuration. The rules of the machine and the moves of the opponent determine the subsequent game moves selected by the LBA. Some initializations may cause the LBA to cycle sometime in the future and thus never pick a next move. Such an initialization does not represent a strategy as defined in Definition 1.2.

We will now express the above concepts in a more formal way. The reader may wish to verify that the formalism does reflect the intuition given above, but the details are really unimportant for understanding the paper. We first define formally a binary relation “ \vdash ” on configurations which says when one configuration follows another. This is a standard automata theoretic method of representing one computation step. We also define a binary relation “ \rightarrow ” to represent a sequence of computation steps that represents the selection of one game move in the super-game.

Definition 2.3. Let $\mathcal{A} = (M_1, M_2, S, \alpha, \Gamma, \tau)$ be a game-playing LBA, let $c_1 = (m_1, s_1, w_1, t_1, x_1)$ and $c_2 = (m_2, s_2, w_2, t_2, x_2)$ be configurations of \mathcal{A} , and let $\alpha(m_1, s_1, t_1) = (s_3, t_3, \mathbf{D})$. Then we write $c_1 \vdash c_2$ if and only if the following conditions hold:

- 1) $s_2 = s_3$
- 2) if $\alpha(s_2) = \text{CONTINUE}$, then $m_2 = m_1$
- 3) $w_1 t_3 x_1 = w_2 t_2 x_2$
- 4) if $\mathbf{D} = \mathbf{R}$ then $|w_2| = |w_1| + 1$; if $\mathbf{D} = \mathbf{L}$ then $|w_2| = |w_1| - 1$; $\mathbf{D} = \mathbf{R}$ then $w_2 = w_1$.

If c_1 and c_2 are output configurations, we write $C_1 \rightarrow C_2$ if and only if $C_1 \vdash C_2$ or there is a sequence of intermediate configurations $d_1 \dots d_k$ such that $c_1 \vdash d_1 \vdash \dots \vdash d_k \vdash c_2$.

■

Lemma 2.4. Let $\mathcal{A} = (M_1, M_2, S, \alpha, \Gamma, \tau)$ be a game-playing LBA. For integer b , let \mathcal{C}_b be the set of configurations of \mathcal{A} such that $(m, s, w, t, x) \in \mathcal{C}_b$ if and only if $|wtx| = b$ (i.e. the length of the tape is b). Then there exists constants K and H such that $|\mathcal{C}_b| \leq 2^{Kb+H}$ for all b . Constant K depends only on the size of Γ . For binary \mathcal{A} , K is independent of \mathcal{A} . ■

Proof. The number of configurations is exactly $|M_2| \cdot |S| \cdot b \cdot (|\Gamma| - 2)^{b-2}$ because b is the number of head positions and $(|\Gamma| - 2)^{b-2}$ is the number of tapes of length b beginning with $\$$, ending with ϕ , and having symbols from $\Gamma - \{\$, \phi\}$ in the middle. For a given \mathcal{A} , $|M_2|$, $|S|$, and $|\Gamma|$ are constants. Because $\log_2 b \leq 2b + 1$, the inequality and claims about K follow. ■

This lemma has both intuitive and technical importance. Intuitively, an LBA really has three methods of remembering information, namely by the contents of the tape squares, the position of the head on the tape, and the state of its finite-state control. However the lemma shows that the number of tape squares b is the dominant contribution, and thus makes a reasonable measure of complexity. The technical importance is that amount of memory needed by the opposing automaton

to remember a configuration of \mathcal{A} is linear in b . This is a key consideration in the main theorem.

Lemma 2.5. Let $\mathcal{A} = (M_1, M_2, S, \alpha, \Gamma, t)$ be a game-playing LBA. There exist numbers K and H such that, for all integers b and all output configuration $c = (m, s, w, t, x)$ with $|wtx| = b$, there exists an output configuration c' such that $c \rightarrow c'$ if and only if there are zero or more intermediate configurations $d_1 \dots d_k$ for some $k < 2^{Kb+H}$ such that $c \vdash d_1 \vdash \dots \vdash d_k \vdash c'$.

Proof. The tape size b does not change as the LBA changes from configuration to configuration thanks to part 3 of Definition 2.3. This definition also requires the existence of $d_1 \dots d_k$ for some $k \geq 0$. Letting K and H be as in Lemma 2.4, we know that if $k \geq 2^{Kb+H}$, we must have $d_i = d_j$ for some $i \neq j$. But in this case, the sequence of intermediate configurations which follows c must be an infinite sequence which repeats the segment $d_i \vdash \dots \vdash d_j$ and so no c' exists. ■

The importance of this lemma is that cycling can be detected by counting instead of remembering configurations. If we find that 2^{Kb+H} steps have taken place without going through an output configuration, we know the machine is cycling. The number of bits required to do this counting is $Kb + H$, a linear function of b .

A game-playing LBA must be initialized before it can execute a strategy. We now give a formal definition of an “initialization”:

Definition 2.6. If $\mathcal{A} = (M_1, M_2, S, \alpha, \Gamma, \tau)$ is a game-playing LBA, we say that $\mathcal{I} = (s, w, t, x)$ is an *initialization* for \mathcal{A} if and only if s, w, t , and x satisfy conditions 2 through 5 of Definition 2.2 and $\alpha(s)$ is in M .

We now relate the initialized LBA to potential output/input sequences. Those consistent with the initialization will be called “matched”:

Definition 2.7. If $\mathcal{A} = (M_1, M_2, S, \alpha, \Gamma, \tau)$ is a game-playing LBA, $\mathcal{I} = (s, w, t, x)$ is an initialization for \mathcal{A} , and $(m_i, m'_i) \dots (m_k, m'_k)$ for some k is a sequence of pairs from $M_1 \times M_2$, we say that \mathcal{I} is *matched* to the sequence if and only if there is a sequence of k output configurations $c_i = (\bar{m}_i, s_i, w_i, t + i, x_i)$ for \mathcal{A} such that $\bar{m}_i = m'_i$ and $\alpha(s_i) = m_i$ for $1 \leq i \leq k$; $s_1 = s$, $w_1 = w$, $t_1 = t$, and $x_1 = x$; and $c_i \rightarrow c_{i+1}$ for

$1 \leq i < k$.

Definition 2.8. Let \mathcal{A}_1 and \mathcal{A}_2 be game-playing LBAs with output set M_1 and input set M_2 . We say that initialization \mathcal{I}_1 and \mathcal{I}_2 *behave equivalently* if and only if, for all k and all sequences $(m_1, m'_1) \dots (m_k, m'_k)$, \mathcal{I}_1 is matched to the sequence if and only if \mathcal{I}_2 is matched to the sequence.

If \mathcal{I}_1 and \mathcal{I}_2 behave equivalently, then they cannot be distinguished by the opponent and they define the same strategy if they do define a strategy. We now want to relate LBA to specific games.

Definition 2.9. If $G = (M_1, M_2, \pi)$ is a game, we say that a game-playing LBA \mathcal{A} is *for player I* if the set of input moves of \mathcal{A} is M_2 and the set of output moves is M_1 . We say \mathcal{A} is *for player II* if the set of input moves is M_1 and the set of output moves is M_2 .

We now give a definition to relate the game-playing LBA to the outcome of a game.

Definition 2.10. Let $G = (M_I, M_{II}, \pi)$ be a game, let $\mathcal{A}_I = (M_I, M_{II}, S, \alpha, \Gamma, \tau)$ be a game-playing LBA for player I. Let (m_i, m'_i) for $i \geq 1$ be a sequence of pairs from $M_I \times M_{II}$. We say that initialization \mathcal{I}_I for \mathcal{A}_I **matches** the sequence if \mathcal{I}_I matches $(m_i, m'_i) \dots (m_k, m'_k)$ for all k . If instead we have $\mathcal{A}_{II} = (M_{II}, M_I, S, \alpha, \Gamma, \tau)$ for player II, we say that initialization \mathcal{I}_{II} for \mathcal{A}_{II} **matches** $(m_i, m'_i) \dots (m_k, m'_k)$ for all k .

Proposition 2.11. For a given game $G = (M_I, M_{II}, \pi)$, given game-playing LBAs \mathcal{A}_I and \mathcal{A}_{II} for players I and II, and given initializations \mathcal{I}_I and \mathcal{I}_{II} for \mathcal{A}_I and \mathcal{A}_{II} , then there is at most one sequence (m_i, m'_i) for $i \geq 1$ matched to both \mathcal{I}_I and \mathcal{I}_{II} . We call this sequence the *result* of \mathcal{I}_I and \mathcal{I}_{II} .

Proof. There is only one possibility for (m_1, m'_1) , namely $(\alpha_I(s_I), \alpha_{II}(s_{II}))$ where s_I and s_{II} are the states of \mathcal{I}_I and \mathcal{I}_{II} . The result follows by induction. ■

We can now summarize our model more precisely. For a given game, players I and II will each have a game-playing LBA. The LBA for player I will input moves of player II and output moves of player I. The LBA for player II does the reverse. It

will turn out that it doesn't matter much whether the players have knowledge of each others LBA, or even if one player designs his LBA with full knowledge of the LBA of the other player.

To play the super-game, each player initializes his machine to be in some state, to have a certain sequence on its tape, and to have the head of the machine on some tape symbol. If no looping occurs, the two LBAs together produce an infinite result sequence (m_i, m'_i) for $i \geq 1$, a unique sequence which is matched to the two initializations chosen by the respective players. It is assumed that the players have picked initializations which cannot loop since initializations which can loop do not define strategies in the sense of Definition 1.2.

Because the LBA does not change the length of the information written on its tape, the number of configurations it can enter is bounded (see Lemma 2.4) by a function of the size of the initialized tape. Thus the LBA behaves like a finite state machine and the discussion after Definition 1.2 applies. A well-defined average payoff results whenever two game-playing LBA play a super-game.

3 Universal LBA

We will assume throughout this section that all LBAs under discussion have some fixed set M_1 as output moves and some fixed set M_2 as input moves. Such an LBA could be used either by player I in some game (M_1, M_2, π) or by player II in some games (M_2, M_1, π') . Because the mathematical objects needed to define these LBAs for this game all have finite size (Definition 2.1) one can write down a complete description of such a machine. Such a description might include a list of six-tuples from $M_2 \times S \times \Gamma \times S \times \Gamma \times \{\mathbf{L}, \mathbf{R}, \mathbf{S}\}$ to describe the transition function. This can be done conveniently with a fixed small set of symbols, adequate for describing any LBA, if we use strings of symbols (say strings of 0's and 1's) to represent state or tape symbols. For example, the tuple $(m, 010, 11, 110, 00, \mathbf{R})$ would mean "when in state 010 with symbol 11 under the tape head and the last move of the opponent was

m, change to state 1101, replace the tape symbol with 00, and move the head one square to the right”. For a given LBA, this can be done in such a way that all states are represented by strings of the same length and all tape symbols are represented by strings of the same length.

Now suppose we have the description of an LBA together with an initialization. From this information, one can compute the sequence of game moves the LBA will produce in response to any sequence of moves of the opponent. More importantly, this computation itself can be performed by an LBA. We call such an LBA a “universal LBA” because it can implement any strategy that can be implemented at all on an LBA. This is a direct analogy with the concept of a “universal Turing machine”.

Theorem 3.1. Consider LBA with fixed output set M_1 and fixed input set M_2 . There exists a game-playing LBA \mathcal{U} (called a **universal** game-playing LBA) which, when initialized with a description of some other game-playing LBA \mathcal{A} and a description of an initialization of \mathcal{I} of \mathcal{A} , the initialization of \mathcal{U} behaves equivalently to \mathcal{I} . Furthermore, for a given \mathcal{A} , there exist constants A and B such that the number of tape squares used by \mathcal{U} is no greater than $A \cdot b + B$ where b is the number of tape squares used by \mathcal{A} .

Proof. The LBA \mathcal{A} can be described as in the discussion prior to the theorem. The initial state and tape can be represented in the obvious ways using the same state names and tape symbol names as in the description. A special symbol can be inserted into the tape description to mark the head position. The tape description must be delimited with surrogate endmarkers since $\$$ and \notin must be reserved to mark the ends of the tape presented to \mathcal{U} . Given a description of the state and tape of one configuration, this description can be changed into a description of the next configuration by replacing the string representing the old state with the string representing the new, replacing the string representing the tape symbol under the head with a string representing the new tape symbol, and rearranging symbols so as to represent the new head position. The next configuration for \mathcal{A} can occupy the same tape squares as the original configuration because all the states are represented

by strings of the same size and all tape symbols are represented by strings of the same size.

As is customary in Turing machine proofs, we omit giving full details. In this case, details omitted include the method used to look-up the appropriate six-tuple in the description or the method used for changing the configuration description once the appropriate tuple has been located. For the uninitiated, we note that these methods require a few additional tape symbols to mark tape positions while the head moves back and forth between the machine description and the configuration. States are used to remember finite amounts of information while the head moves back and forth.

Regarding the constants A and B , B represents the space needed to write down the description of \mathcal{A} and the configuration state. Constant A represents the number of tape squares of the universal machine needed to store a string representing one tape symbol of \mathcal{A} . ■

A universal machine can be thought of as a general-purpose computer and the machine descriptions on its tape as programs. The theorem then says there is a general-purpose LBA that can be programmed to carry out any task that can be carried on any other LBA. The idea that arbitrary LBAs can be simulated in linear space (specifically $A \cdot b + B$) is all that is needed for our main result. However, we also want to derive some results about memory bits as a complexity measure. To do this, we need a corresponding result for binary LBA:

Theorem 3.2. Consider LBA with fixed output set M_1 and fixed input set M_2 . There exists a binary game-playing LBA \mathcal{U} such that, given some other binary game-playing LBA \mathcal{A} and given an initialization of \mathcal{I} of \mathcal{A} , there is an initialization of \mathcal{U} which behaves equivalently to \mathcal{I} . Furthermore, for each \mathcal{A} , there is a constant $K_{\mathcal{A}}$ such that the number of bits (tape squares) used by \mathcal{U} is no more than $b + K_{\mathcal{A}}$ where b is the number of bits used by \mathcal{A} .

Proof. The universal machine constructed in the proof of Theorem 3.1 can be made binary. To do this, first choose some sufficient length k and encode each tape

symbol (other than \$ and \emptyset) of the universal machine as a sequence of k bits (symbols from $\{0,1\}$). Then modify the state set so that it can read k tape squares and behave as if the original machine had read the corresponding symbol of its tape symbol set. This binary machine satisfies the first part of the theorem but not the part about the bound $b + B_{\mathcal{A}}$.

The difficulty is that the above encoding will represent the tape symbols 0 and 1 of \mathcal{A} by sequences of length k and so the bound would be $k \cdot b + B_{\mathcal{A}}$. To get around this, we adopt the encoding suggested above for encoding the description of \mathcal{A} and remembering the state of the current configuration of \mathcal{A} . However, the tape symbols of the current configuration will be stored unencoded as single symbols 0 or 1. This means that \mathcal{U} , through use of its state set, must be able to distinguish which zeros and ones are used to encode the description of \mathcal{A} and which zeros and ones are actual tape symbols from the current configuration of \mathcal{A} .

There are several methods of making the required discrimination between the two usages of zeros and ones. One method is to keep the description of \mathcal{A} in the middle of the tape adjacent to the current head position. Thus the LBA can look up the current tape symbol simply by moving to a tape square adjacent to the description. When \mathcal{A} shifts its head, \mathcal{U} may have to shift the entire description of \mathcal{A} . The machine \mathcal{U} can be designed to allow the tape symbol under the head to be on either side of the description. This flexibility is needed so that the description can be kept to the right of the left endmarker and to the left of the right endmarker.

Following the above method, the amount of tape required to simulate \mathcal{A} is some constant amount $K_{\mathcal{A}}$ required to store the description of \mathcal{A} plus the number of tape squares b required to store the contents of the tape of \mathcal{A} . Thus the total requirement is $b + K_{\mathcal{A}}$. ■

The above construction has an advantage over some other methods of representing configurations.

Corollary 3.3. There is an LBA \mathcal{U} satisfying Theorem 3.2 with the additional property that there exists a constant $H_{\mathcal{A}}$ such that the number of transitions taken by

\mathcal{U} to compute a given game move is no more than $H_{\mathcal{A}}$ times the number of transitions of \mathcal{A} .

Proof. To simulate one transition of \mathcal{A} , the head of \mathcal{U} constructed in the proof of the theorem is confined to $K_{\mathcal{A}} + 2$ tape squares and therefore executes one of a bounded number action sequences. ■

4 Bits as a Complexity Measure

We now look at the suitability of counting tape squares on a binary LBA as a method of measuring the complexity of finite-state strategies. As discussed in the introduction, this concept is imprecise unless a computing machine is specified.

Definition 4.1. Let M_1 and M_2 be given input and output move sets and let \mathcal{A} be a binary LBA which uses these sets. For any strategy σ , let $B_{\mathcal{A}}(\sigma)$ be the minimum number of tape squares needed so that \mathcal{A} , with the squares properly initialized, carries out σ . (We do not count the endmarker squares as part of this number.) $B_{\mathcal{A}}(\sigma)$ is undefined if no initialization carries out σ .

This concept has a weakness in that the measure depends on which LBA is used. Furthermore, any finite-state strategy can be carried out with zero tape squares on some LBA, namely an LBA with a set of states and transitions which imitates the finite-state automata. Nevertheless, there is a choice of machine which does give a fairly satisfactory measure:

Theorem 4.2. Given input and output move sets M_1 and M_2 , there is an LBA \mathcal{U} using M_1 and M_2 such that \mathcal{U} can carry out any finite-state strategy and such that, for any other LBA \mathcal{A} using M_1 and M_2 ,

$$B_{\mathcal{U}}(\sigma) \leq B_{\mathcal{A}}(\sigma) + K_{\mathcal{A}}$$

for all finite-state strategies σ .

Proof. The universal binary \mathcal{U} from Theorem 3.2 is such an LBA. We know that it can imitate any finite-state automaton since a finite-state automaton is, in effect,

a special case of an LBA. Moreover, the $K_{\mathcal{A}}$ provided by the theorem is the same $K_{\mathcal{A}}$ we need here. ■

The bit-counting approach does not permit us to make precise statements such as “strategy X takes 2137 bits” unless it is with respect to some carefully chosen but still very arbitrary LBA. However, Theorem 4.2 tells us that the certain measures, namely those based on universal LBAs which satisfy Theorem 3.2, are fairly satisfactory. For any pair of such measures, there is a constant which bounds the difference between them. Furthermore, these universal LBAs can implement any finite-state strategy with “near optimal efficiency” in the sense that no other LBA can produce measures which are more than some constant smaller.

It might seem that we are being arbitrary in selecting the LBA as our model of a memory-bounded computing device, and that some other device might give radically different measures. However, the technique of one universal machine simulating all others can be extended so that one LBA simulates all members of a broad class of memory-bounded devices. Any memory-bounded computing device is likely to have a finite description and an easily computed rule for accessing its memory and changing states. Using these facts and some mapping of memory locations into consecutive integers, it should be easy to build an LBA which takes a computing device description from some class and simulates the device using a bounded tape. This new “universal LBA” will give a measure as good as any produced on any of the computing devices in question. Because this measure is now achieved by an LBA, it can be achieved by \mathcal{A} of Theorem 4.2. In effect, the inequality $B_U(\sigma) \leq B_{\mathcal{A}}(\sigma) + K_{\mathcal{A}}$ applies even when \mathcal{A} is allowed to range over a much wider class of memory-bounded computing devices.

The LBA model is not so natural if one wants to take into account the number of intermediate configurations the device goes through to produce the next game move. If we want to control the computation time in addition to computing space, then other models should be considered.

5 Procedure for Winning

In this section we give a procedure whereby player I can “defeat” player II when the procedure is given a description \mathcal{A}_{II} of player II’s binary game-playing LBA, the number of tape squares b player II is using, and a parameter ϵ . The procedure will pick a series of moves for player I, depending on the moves of player II, such that player I wins at least the amount $v = \min_{m' \in M_{II}} \max_{m \in M_I} \pi(m, m')$ on at least $(1 - \epsilon)t$ times out of t plays of the game for sufficiently large t . This is done by matching most of player II’s moves with the best response at least $(1 - \epsilon)t$ times. Since player II can guarantee at least v on each move with the strategy of always playing a move m' which minimizes $\max \pi(m, m')$, v is the most player I could guarantee against memory-bounded strategies.

In the next section, we will discuss implementing the procedure on a binary game-playing LBA and discuss the number of tape squares (the space) that will be needed to execute the procedure. It will turn out that the space is linear in b . We will be giving observations on memory requirements at the end of this section.

For a given \mathcal{A}_{II} , b , and ϵ , one can compute a number D such that the number of configurations of \mathcal{A}_{II} using b tape squares is less than 2^D . Let $T = \lceil D/\epsilon \rceil$. For fixed \mathcal{A}_{II} and ϵ , Lemma 2.4 tells us that D and T are linear in b .

The procedure will treat the sequence of plays as being divided into a sequence of segments of size T . During any given segment, the procedure will produce the right response at least $(1 - \epsilon)T$ times and the wrong response at most $D(= \epsilon T)$ times. After the segment is completed, the procedure forgets the information accumulated during that period and begins from scratch as if the first move of the segment were the first move of the game.

During the play of a segment, the procedure will remember the sequence of moves selected by each player. At the end of the segment, a sequence of T moves must be remembered.

To select a move, the procedure considers each possible output configuration of

\mathcal{A}_{II} with b tape squares and evaluates that configuration as a possible configuration of \mathcal{A}_{II} at the beginning of the segment. The evaluation consists of simulating that configuration using the sequence of moves that have been recorded for the segment. The simulation can have three outcomes:

1. A situation is reached where the move indicated by the simulation does not match the actual move made by player II.
2. The simulation cycles through a sequence of intermediate configurations.
3. The simulation is in agreement with the history of the segment and it is determined that some move m' will be made by \mathcal{A}_{II} if \mathcal{A}_{II} really was in the test configuration of the beginning of the segment.

In order to accommodate outcome 2, the procedure will count the number of consecutive intermediate configurations that \mathcal{A}_{II} goes through. If that number exceeds the total number of configurations, the procedure knows that outcome 2 has been reached and the simulation can be halted.

The procedure has an array COUNT indexed by M_1 and COUNT(m) is set equal to zero for all m in M_1 at the beginning of the testing. Whenever situation 3 is reached, the procedure increments $COUNT(r(m'))$ where m' is the next move of \mathcal{A}_{II} predicted by the simulation. Thus after the testing is completed, the procedure knows how many of the initial configurations suggest each m in M_1 as the best next response to \mathcal{A}_{II} . The procedure selects an m which has the largest COUNT(m) as the next game move for player I.

We now prove a lemma which says the procedure plays as well as we have claimed.

Lemma 5.1. During each segment of plays, the procedure described above produces a sequence of outcomes (m_i, m'_i) for $1 \leq i \leq T$ where $(1 - \epsilon)T$ of these pairs satisfy $m_i = r(m'_i)$.

Proof. For each i , let \mathcal{C}_i be the set of configurations of \mathcal{A}_{II} which have outcome 3 during the evaluation phase. A configuration c is in \mathcal{C}_{i+1} if and only if c is in \mathcal{C}_i

and m'_i is the move player II would make from configuration c in response to moves $m_1 \dots m_{i-1}$ of player I. If $m_i \neq r(m'_i)$, then the size of \mathcal{C}_{i+1} can be no more than half the size of \mathcal{C}_i because m_i was chosen to be the optimal response to at least as many configurations as any other move including move $r(m'_i)$. Thus $m_i \neq r(m'_i)$ implies $|\mathcal{C}_{i+1}| \leq |\mathcal{C}_i|/2$.

Now $|\mathcal{C}_i|$ is always at least one since all \mathcal{C}_i contain the configuration that \mathcal{A}_{II} was actually in at the beginning of the segment. Furthermore, \mathcal{C}_1 , the set of all output configurations, has no more than 2^D members by definition of D. But a set of size 2^D cannot be cut by at least one-half more than D time without becoming empty. Thus the number of (m_i, m'_i) with $m_i \neq r(m'_i)$ is no more than D. By construction, $T \geq D/\epsilon$ and the result follows. ■

The memory needs of the procedure can be summarized as follows:

- A. Store the description of \mathcal{A}_{II} .
- B. Save T pairs from $M_1 \times M_2$.
- C. Save the last configuration of \mathcal{A}_{II} to be tested.
- D. Remember the current configuration of \mathcal{A}_{II} in the simulation.
- E. Counting the number of consecutive intermediate configurations in the simulation.
- F. For each m in M_1 , memory for COUNT(m).

In the next section, we will analyze how each of these requirements impacts the implementation of the procedure on an LBA.

6 Main Result

To state the main result, we need a formal definition of “defeat”:

Definition 6.1. Let $G = (M_1, M_2, \pi)$ be a game, let \mathcal{A}_I be a game-playing LBA for player I, and let \mathcal{A}_{II} be a game-playing LBA for player II. We say that initialization \mathcal{I}_I of \mathcal{A}_I *defeats* initialization \mathcal{I}_{II} of \mathcal{A}_{II} *within* $\epsilon > 0$ if and only if the result (if any) for \mathcal{I}_I and \mathcal{I}_{II} , namely (m_i, m'_i) for $i \geq 1$, satisfies

$$\lim_{k \rightarrow \infty} |\{(m_i, m'_i) \mid i \leq k \text{ and } m_i \neq r(m'_i)\}| / k \leq \epsilon.$$

We know that if the result sequence exists, the limit exists because of the cyclic behavior of memory-bounded machine play. The result sequence will fail to exist if one machine starts cycling in its effort to compute a next move.

Theorem 6.2. Let a game $G = (M_1, M_2, \pi)$ be given. There is a game-playing LBA \mathcal{A}_I for player I such that, for any game-playing LBA \mathcal{A}_{II} for player II and $\epsilon > 0$, there exist constants A and B such that for any integer b, \mathcal{A}_I has an initialization with $A \cdot b + B$ tape squares such that the initialization defeats within ϵ any initialization of \mathcal{A}_{II} using b or less tape squares. Furthermore, the constant A depends only on ϵ and the size of the tape alphabet of \mathcal{A}_{II} .

Proof. \mathcal{A}_I will be an LBA which performs the procedure of the previous section. We take it as obvious that this can be done, even though the details of the automaton are quite complex. The initialization of \mathcal{A}_I includes a description of \mathcal{A}_{II} and a partition of the tape into zones, each of which will store information as listed at the end of the previous section. The sizes of these zones are determined by D and T which are derived from \mathcal{A}_{II} , ϵ , and b. The initialization will defeat within ϵ the initialization of \mathcal{A}_{II} because of Lemma 5.1.

We now derive a bound on the number of memory bits which are sufficient to carry out the algorithm. The fact that this bound is linear in b follows mainly from Lemma 2.4 which says D can be picked to satisfy $D \leq K \cdot b + H$ where K and H depend only on \mathcal{A}_{II} . To derive the bound, we find a bound for each use of memory listed at the end of Section 5.

Use A: Storing the description of \mathcal{A}_{II} requires a number of bits which depend only on \mathcal{A}_{II} .

Use B. Remembering the sequence of T output/input pairs requires $\log_2(|M_1 \times M_2|) \times T$ bits. Since $T = \lceil D/\epsilon \rceil$, $T \approx (K/\epsilon) \cdot b + H/\epsilon$ and this use is linear in b since K/ϵ and H/ϵ are constants which depend only on \mathcal{A}_{II} and ϵ .

Uses C, D, E: Each of these requires D bits.

Use F: This requires $|M_1| \cdot D$ bits, still a linear function of b.

Each of the above bounds satisfies the theorem and so must their sum representing the total memory requirements. Thus the memory requirement is linear in b.

To see that \mathcal{A} depends only on ϵ and the size of the tape alphabet, recall that K of Lemma 2.4 depends only on the size of the tape alphabet. Use A does not contribute to constant A. Use B contributes $\log_2(|M_1 \times M_n|) \cdot K/\epsilon$. But $|M_1 \times M_2|$ depends only on the game G and not which \mathcal{A}_{II} chosen to play the game. Thus this contribution depends only on the tape alphabet and ϵ .

Uses C, D, E contribute only K. Use F contributes $|M_1| \cdot K$ but $|M_1|$ depends on G and not on \mathcal{A}_{II} . ■

We now put the main result in terms of binary LBAs:

Theorem 6.3. Let a game $G = (M_1, M_2, \pi)$ be given. There is a binary game-playing LBA \mathcal{A}_I for player I such that, for any binary game-playing LBA \mathcal{A}_{II} for player II and $\epsilon > 0$, there exists a constant E depending only on ϵ and a constant M depending only on \mathcal{A}_{II} such that for any integer b, \mathcal{A}_I has an initialization of size $E \cdot (b + M)$ such that the initialization defeats within ϵ any initialization for \mathcal{A}_{II} using b or less tape squares.

Proof. Any \mathcal{A}_I can be made binary with only a constant effect on the number of tape squares needed to implement a strategy. Standard techniques to do this were discussed in the proof of Theorem 3.2. Restricting \mathcal{A}_{II} to binary LBAs makes the constant K in the proof (K from Lemma 4.2) become independent of \mathcal{A}_{II} . Thus in the expression $A \cdot b + B$, A becomes independent of \mathcal{A}_{II} . Constant B is of the sum of H/ϵ plus other terms which do not depend on ϵ and thus $A \cdot b + B \leq E \cdot (b + M)$ for appropriate E and M. ■

7 Conclusions

We have proposed that the number of bits needed in a memory-bounded computing device as a measure of the complexity of a strategy. To be specific, we have chosen the linear-bounded automaton (LBA) as our computer model. However, the bits needed on this device is not substantially different than on other automata theory models.

We believe that bit counting is a more intuitive measure than state counting. The contrast between the measures was illustrated with strategies B and C in the introduction. From the perspective of bit counting, both these seem about the same. From the state counting perspective, B is exponentially harder than C.

Bit counting is a complexity measure in the style of Kolmogorov complexity for finite sequences, and these measures have similar properties. Although no exact measure is produced by this theory, there are measures defined by “universal” devices which are always smaller, except for an additive constant, when compared with another such measurement.

The main result says that player I needs only an initialization of $A \cdot b + B$ on a certain LBA to defeat any initialization of b bits on some arbitrary machine of player 2 (where A and B depend on player II’s machine and the tolerance ϵ for a wrong move). This contrasts with the state-counting result of [B-P] which says that an exponential number of states is required for one machine to defeat another.

The strategy we use to establish our main result requires a number of computation steps between two game moves which is exponential in b . This means that the strategy is not of practical use. This suggests that computation “time”, the number of steps between moves, should somehow also be taken into account. A complexity based on both time and memory on LBAs will retain the key properties of memory measurement because of Corollary 3.3. However changing from an LBA to another computing device could now have significant effects on the complexity measure. In any case, we know very little about how many additional bits are needed to defeat a machine in a “reasonable” number of steps.

More generally, it seems that computation time should be taken into account when modeling “bounded rationally”. In the real world, decisions must be made in a timely manner and this imposes a limitation just as surely as does limits on computer hardware or the memory limitations of the human brain.

References

- [Au] R. Aumann, Survey of repeated games, in *Essays in Game Theory and Mathematical Economics in Honor of Oskar Morgenstern*, Bibliographisches Institut Mannheim/Wien/Zurich, 1981
- [B-P] Elchanan Ben-Porath, Repeated Games with Finite Automata, based on M.S. thesis, Institute for Mathematical Studies in the Social Sciences, Stanford University, 1986.
- [KS] E. Kalai and W. Stanford Finite rationality and interpersonal complexity in repeated games, *Econometrica*, 1986.
- [Ko] A.N. Kolmogorov, Three approaches to the quantitative definition of information, *Problems in Information Transmission* 1(1), pp. 1-7, 1965.
- [Le] L.A. Levin, Laws of information conservation (non-growth) and aspects of the foundation of probability theory, *Problems in Information Transformation* 10, pp. 206-210, 1974.
- [Ne] A. Neyman, Bounded Complexity Justifies Cooperation in the Finitely Repeated Prisoners’ Dilemma, *Economic Letters* 19, pp. 227-229, 1985.
- [Ru] A. Rubenstein, Finite automata play the repeated prisoners’ dilemma, *Journal of Economic Theory*, 39, 83-96, 1986.
- [So] R.J. Solomonoff, A formal theory of inductive inference, Part 1 and Part 2, *Information and Control* 7, pp. 1-22, 224-254, 1964.