Time-Space Tradeoffs for the Memory Game

Dartmouth Computer Science Technical Report TR2018-860

Yining Chen[*]
Advisor: Amit Chakrabarti[†]

Abstract

A single-player game of MEMORY is played with n distinct pairs of cards, with the cards in each pair bearing identical pictures. The cards are laid face-down. A move consists of revealing two cards, chosen adaptively. If these cards match, *i.e.*, they bear the same picture, they are removed from play; otherwise, they are turned back to face down. The object of the game is to clear all cards while minimizing the number of moves. Past works have thoroughly studied the expected number of moves required, assuming optimal play by a player has that has *perfect* memory. In this work, we study the MEMORY game in a *space-bounded* setting.

We prove two time-space tradeoff lower bounds on algorithms (strategies for the player) that clear all cards in T moves while using at most S bits of memory. First, in a simple model where the pictures on the cards may only be compared for equality, we prove that $ST = \Omega(n^2 \log n)$. This is tight: it is easy to achieve $ST = O(n^2 \log n)$ essentially everywhere on this tradeoff curve. Second, in a more general model that allows arbitrary computations, we prove that $ST^2 = \Omega(n^3)$. We prove this latter tradeoff by modeling strategies as branching programs and extending a classic counting argument of Borodin and Cook with a novel probabilistic argument. We conjecture that the stronger tradeoff $ST = \widetilde{\Omega}(n^2)$ in fact holds even in this general model.

Keywords: time-space tradeoffs; branching programs; matchings; probabilistic method

[*]Department of Computer Science, Dartmouth College, Hanover, NH.
[†]Department of Computer Science, Dartmouth College, Hanover, NH. Supported in part by NSF under award CCF-1650992.

1 Background

1.1 Finite Function Complexity

While algorithms give upper bounds on the computational resources needed to solve problems, complexity theory seeks to prove that some problems cannot be solved efficiently by *any* algorithm. Two most important computation resources are time and space. Time is usually defined as the numbers of steps an algorithm takes, and space defined as the number of bits required to compute the output for a given input. Specific problems' time and space complexities depend on the computational model. The choice of computational model restricts the class of potential algorithms. When we analyze the runtime and space usage of well-known algorithms such as depth-first search or Dijkstra's shortest-path algorithm, the computational model is usually implicitly the Random-Access Machine (RAM). Both RAM and the Turing machine are uniform computation models, meaning that the same machine is used for inputs of all sizes. Since algorithms can use unexpected strategies, it is usually hard to prove lower bounds on the computational resources required to solve specific problems in the most general computation models. One way to make progress on the lower bounds is to limit the computation model in order to limit the class of potential algorithms. The hope is that the lower bounds obtained in restricted models will eventually be generalized to more powerful models.

We now introduce four restricted computational models, for which complexity theorists have proven lower bounds on computing finite functions: Boolean circuits, formulas, decision trees and branching programs. Let n be a finite positive integer, and X, Y be finite sets. A *finite function* is a partial function $f : X' \to Y$ defined on only finitely many n-tuples $X' \subseteq X^n$. A Boolean function $f : \{0,1\}^n \to \{0,1\}$ is an example of finite function. See Boppana and Sipser's survey [BS90] on the complexity of Boolean functions for Boolean circuits, formulas, and branching programs. Unlike RAM and the Turing machine, these models are nonuniform, meaning that a different algorithm can be used for each input size n.

A *Boolean circuit* is a simplified model of the chips used to make modern computers. It is directed acyclic graph. The input nodes have indegree 0 and are labeled with a variable x_i or with a constant 0 or 1. The nodes with indegree $k > 0$ are called *gates* and are labeled with a Boolean function on its k inputs. The collection of Boolean functions used as gates is called a *basis*. For example, the DeMorgan basis $D = \{NOT(\neg), AND(\vee), OR(\wedge)\}$. One node is the output node. The size of the circuit is the number of gates, and the depth is the maximum distance from an input to the output. Figure 1 is a Boolean circuit that computes XOR on two inputs x_1, x_2.

For a Boolean function f, the size of the smallest circuit computing f is called its circuit complexity, $C(f)$. There has been little progress on general circuit lower bounds: No super-linear circuit lower bound have been proven for any NP problem. Some progress is made for restricted circuit classes, such as bounded

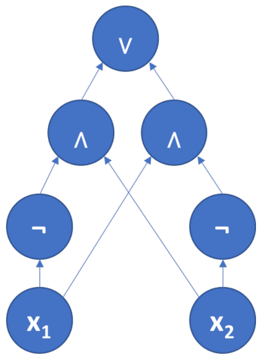


Figure 1: Boolean circuit of XOR

depth circuits, bounded depth circuits with special gates, and monotone circuits. Usually there is a notion of "progress" of the computation, and analysis shows that small circuits cannot achieve the amount of progress required to compute the output from the inputs. See Chapter 14 in Arora and Barak's textbook [AB09] for a survey of results in circuit complexity.

A *Boolean formula* is a special kind of Boolean circuit whose underlying graph is a tree (all gates have outdegree at most 1). Figure 2 is a formula that computes XOR. We can unfold this tree into a Boolean expression consisting of literals (variables or their negations) and Boolean functions: $((\neg x_1) \wedge x_2) \vee (x_1 \wedge (\neg x_2))$. The size of a formula is the number of occurrences of literals in the formula, and the size of the smallest formula that computes f is denoted as the formula complexity $L(f)$. See Section 5 in [BS90] for a survey on lower bounds for formulas with different bases.

A *decision tree* of a Boolean function f is a binary tree whose internal vertices are labeled by variables and leaves are labeled by 0 and 1. Each internal vertex has exactly two outgoing edges, labeled by 0 and 1. Given an assignment to the input variables, the function value is computed by following a path starting at the root, querying the variable designated by the label of the current vertex, following the edge labeled with the variable's value, and going to the next vertex, until we reach a leaf. The output is the label of the leaf vertex. The *decision tree complexity* of f, $D(f)$, is the minimum depth of decision trees computing f. See Chapter 5 in [DK00] for a survey on the decision tree complexity of various Boolean functions.

The definition of decision tree for Boolean functions can be extended to any finite function $f : [X]^n \to Y$. Each node queries a variable x_i and branches into $|X| = R$ subtrees, one for each possible value of x_i. See figure 3 for an example. Besides general decision trees whose nodes directly query variable values, *comparison trees*, *linear decision trees* and *algebraic computation trees* have also been studied. In a comparison tree, each node compares two variables $x_i : x_j$ and branches into three subtrees corresponding to the three cases $x_i < x_j$, $x_i = x_j$, and $x_i > x_j$. See figure 4. One example of comparison tree lower bound is that sorting n variables requires decision trees of height $\Omega(n \log n)$. In a linear comparison tree, each node v queries $\sum_{i=1}^{n} \lambda_i^v x_i : c^v$ and branches into three subtrees $\sum_{i=1}^{n} \lambda_i^v x_i < c^v$, $\sum_{i=1}^{n} \lambda_i^v x_i = c^v$, $\sum_{i=1}^{n} \lambda_i^v x_i > c^v$, where λ_i^v and c^v are arbitrary real numbers. See figure 5. An algebraic computation tree has some nodes performing algebraic operations (multiplying two ancestors, multiplying one ancestor with a constant, or taking square root of an ancestor) and other nodes performing comparison queries (whether an ancestor is greater than 0). Ben-Or shows that any algebraic computation tree that decides whether n inputs variables are distinct has height $\Omega(n \log n)$ [BO83]. The output of decision trees can be single- or multi-valued (such as in sorting), and can be attached to either only the leaf nodes or any edge. In the latter case outputs are collected over the entire computation path from the root to the leaf.

Identical subtrees in a decision tree can be combined. Therefore, We can use a directed acyclic graph instead of a tree to represent a decision process. Such a "decision graph" is called a *branching program*,

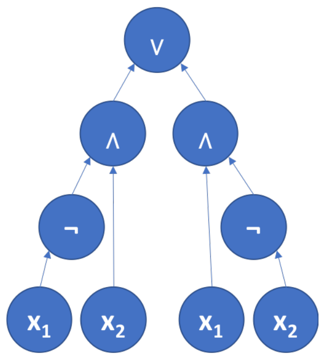


Figure 2: Boolean formula of XOR

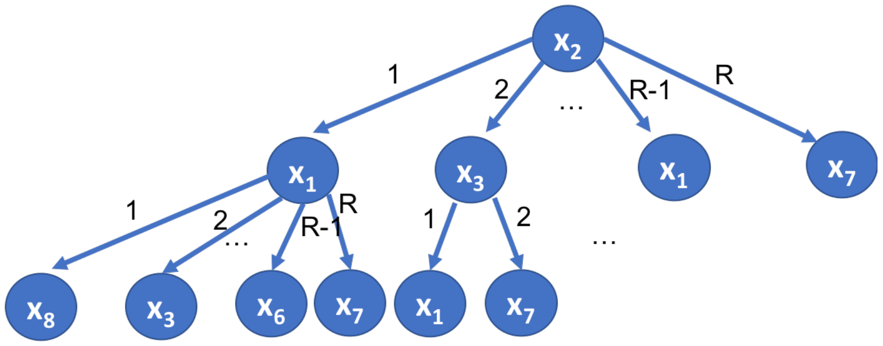


Figure 3: Example of a decision tree

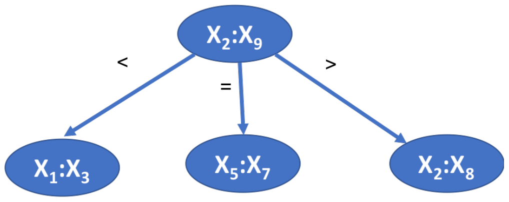


Figure 4: Example of a comparison tree (or a comparison branching program)

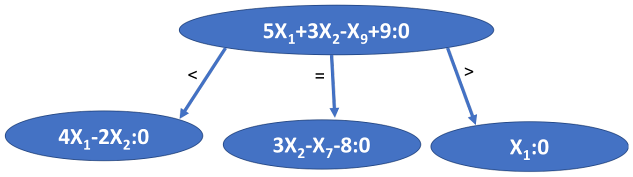


Figure 5: Example of a linear comparison tree (or a linear comparison branching program)

which is the primary model we study. We will introduce the model in detail in section 3.1. The two important complexity measures of a branching program are depth and width. The depth corresponds to the running time of an algorithm and the width corresponds to the space. The branching program is thus a natural model to simultaneously study time and space complexities.

1.2 Concentration of Measure

If we toss a fair coin 1000 times, the number of heads is very likely to be around 500. Although there are many possible outcomes, those that are likely to be observed is usually concentrated in a narrow range. This phenomenon is called concentration of measure. There are many techniques to bound the probability that random variable X takes values that deviate far from its expected value $\mathbb{E}X$. The simplest tail bound is Markov's inequality: For $X \geqslant 0$ and $a > 0$,

$$\Pr[X \geqslant a] \leqslant \frac{\mathbb{E}X}{a}.$$

This bound is quite loose. For example, let X be the number on top when we throw a fair die. Clearly $\mathbb{E}X = 3.5$. Markov's inequality would give us $\Pr[X \geqslant 7] \leqslant 3.5/7 = 1/2$, while this probability is actually 0.

A tighter bound is Chebyshev's inequality:

$$\Pr[|X - \mathbb{E}X| \geqslant a] \leqslant \frac{Var(X)}{a^2}$$

where $a > 0$ and $Var(X)$ is the variance of X. In the example of throwing a fair die, $Var(X) = \sum_{i=1}^{6} (i - 3.5)^2/6 = 35/12$. Chebyshev's inequality gives us $\Pr[|X - 3.5| \geqslant 3.5] \leqslant 35/(12 * 3.5)$, or $\Pr[X \geqslant 7] \leqslant 5/12$.

We will use the Chernoff-Hoeffding bound in our proof. Let Z be the total number of heads when we toss a biased coin n times. Let p be the probability of head in each toss. Then Z is a binomially distributed random variable and can be written as a sum $Z = \sum_{i \in [n]} X_i$, where X_i ($i \in [n]$) is the indicator random variable defined by $X_i = 1$ if the ith toss is head and $X_i = 0$ otherwise. The Chernoff-Hoeffding bound states that, for all reals a, p, with $0 < p < a < 1$,

$$\Pr[Z \geqslant an] \leqslant e^{-n\mathrm{D}(a\|p)}, \tag{1}$$

$$\text{where } \mathrm{D}(a\|p) = a \ln \frac{a}{p} + (1-a) \ln \frac{1-a}{1-p} \tag{2}$$

is the relative entropy of the Bernoulli distribution Bern(a) to Bern(p) (Theorem 1 of Arratia and Gordon [AG89]).

One requirement for applying the Chernoff-Hoefding bound is that $\{X_i : i \in [n]\}$ have to be independent. However, in some dependent situations, the bound can be applicable as is or with some modifications. One circumstance where the bound is salvageable is when $\{X_i : i \in [n]\}$ are negatively associated. Random variables are *negative associated* if for all disjoint subsets $I, J \subseteq [n]$ and all non-decreasing functions f and g,

$$\mathbb{E} f(X_i, i \in I) g(X_j, j \in J) \leqslant \mathbb{E} f(X_i, i \in I) \mathbb{E} g(X_j, j \in J)$$

(Definition 3.1 in [DP09]). Intuitively, conditioned on a subset of indicator random variables taking high values, a disjoint subset take low values.

When $\{X_i : i \in [n]\}$ are negatively associated, Chernoff-Hoeffding bounds can be applied as is to $Z = \sum_{i \in [n]} X_i$. See Chapter 3 in Dubhashi and Panconesi's textbook [DP09] for more on applying Chernoff-Hoeffding bounds in dependent settings.

2 Introduction

The popular children's card game "Memory" (also known as "Concentration") is played with a deck of n pairs of picture cards: the two cards in each pair have the same picture, while two cards taken from two distinct pairs have distinct pictures. The game starts with these cards facing down on a table. A *move* in the game consists of flipping up one card to reveal it, and then flipping up a second card: if the two revealed cards are from the same pair—i.e., they *match*—they are removed from the table, otherwise they are flipped back to face down. The game ends when all cards have been removed. From these basic rules, one can formulate a two-player or multi-player game, but this paper is concerned with the one-player "Solitaire Memory" game, which we shall simply call MEMORY in this paper. The goal of MEMORY is to remove all cards as quickly as possible, i.e., using a minimum number of moves.

Versions of this game have attracted some attention from researchers in the recent past. Alfthan [Alf07] studied strategies for two-player versions of the game. Foerster and Wattenhofer [FW13] studied the solitaire game and derived the first nontrivial upper and lower bounds on the expected number, $T(n)$, of moves required by an optimal strategy. Subsequently, Velleman and Warrington [VW13] proved the very tight result that $T(n) = (3-2\ln 2)n + 7/8 - 2\ln 2 + o(1) \approx 1.61n$. However, all of these works assume that the player(s) have *perfect* memory. Arguably, what makes the children's game Memory interesting is that player(s) will find it hard to remember everything they have learned from previous moves. Thus, from a computational complexity perspective, the most interesting question about MEMORY is: what is the minimum expected number of moves required by a player whose memory capacity is at most S bits?

2.1 Our Results and Techniques

Define an (S,T)-strategy to be one where the player flips at most $T = T(n)$ cards, while using at most $S = S(n)$ bits of memory throughout. Notice that we are measuring "time" in terms of cards flipped (rather than moves made): this will simplify the presentation and is accurate enough for our purposes, since we shall prove asymptotic results. We shall soon make this definition more formal. For now, we note that we can readily obtain deterministic (S,T)-strategies with $ST = \widetilde{O}(n^2)$, for essentially all S between $\widetilde{\Theta}(1)$ and $\widetilde{\Theta}(n)$.[1] We assume that the picture on each card can be represented using $\widetilde{O}(1)$ bits.

The main goal of this work is to study corresponding lower bounds, i.e., time-space tradeoffs. We prove two such tradeoffs. The first applies to a simple model in which the player is deterministic and can do only one thing with the pictures on the cards, namely, compare them for equality. Under this restriction, we show that $ST = \widetilde{\Omega}(n^2)$ is necessary; see Theorem 2. We do this by analyzing an explicit adversarial strategy, using concepts from matching theory. The aforementioned upper bound $ST = \widetilde{O}(n^2)$ is achieved by a strategy that does operate in this simple model, so this lower bound is tight.

Our second tradeoff (and main result) applies to a fully general model, where the player may use randomization and may treat the picture on each card as an integer and perform arbitrary computations with these integers. We prove that every (S,T)-strategy must satisfy $ST^2 = \Omega(n^3)$; see Theorem 3 in Section 5.1. We do this by modeling a strategy for MEMORY as a general branching program (BP): there are $2n$ input variables corresponding to the $2n$ cards; each node in the BP reads a variable (flips a card); and each edge in the BP has the potential to produce output (declare some pairs of cards as matched). To prove the appropriate time-space tradeoff for BPs that successfully play MEMORY, we extend a classic counting argument due to Borodin and Cook [BC82] with a novel probabilistic argument of our own.

[1]The $\widetilde{O}(\cdot)$ and $\widetilde{\Theta}(.)$ notations ignore factors polylogarithmic in n; in this particular instance, the factor ignored is $O(\log n)$.

2.2 Related Work

We have already discussed past work on the Memory game itself [Alf07, FW13, VW13]. As those works do not deal with a space-bounded computational setting, at the technical level they are largely unconnected to our work. The relevant related work is mostly in the area of time-space tradeoffs for branching programs. For a detailed overview of this area and an excellent exposition of many key results, we refer the reader to Chapter 10 of the textbook by Savage [Sav97]. Our work was motivated in part by a goal of furthering our understanding of space-limited algorithms for computing matchings in graphs: MEMORY is a toy version of this much richer problem. Thus, the recent and growing body of work on streaming algorithms and lower bounds for computing matchings in graphs is also relevant: see, e.g., Assadi et al. [AKL17], Crouch and Stubbs [CS14], and the references therein. We discuss this motivation further at the end of the paper, in Section 6.

In a highly influential work, Borodin and Cook [BC82] studied the SORTING problem: given integers $x_1, x_2 \ldots, x_n \in [n^3]$, output the x_i values in sorted order.[2] They devised a counting-based method for deriving time-space lower bounds for branching programs, using which they proved the tradeoff $ST = \Omega(n^2/\log n)$ for SORTING. Their method's essential feature is that it divides time into "stages" and applies a counting argument to each stage to argue that, if the branching program is "too small," then none of the stages can make enough progress on sufficiently many inputs. We shall see this overall scheme in the proof of our main theorem. The Borodin-Cook method has been applied to many other problems, including matrix multiplication over a finite field [Yes84], generalized string matching [Abr87], Boolean matrix multiplication [Abr90], calculating universal hash functions [MNT93], and UNIQUE-ELEMENTS [Bea91].

The Borodin-Cook method is described in detail in Savage's book [Sav97]; in particular, Theorem 10.11.1 in that book gives a very general version of the method. In the terminology of that theorem, the conditions under which this method applies are captured by a property of functions called $(\phi, \lambda, \mu, \nu, \tau)$-distinguishability. The function that describes the desired output of the game MEMORY is not $(\phi, \lambda, \mu, \nu, \tau)$-distinguishable for any positive constant ν, so this method does not apply as is to our problem.

It is worth comparing the quality of our branching program lower bound to known results on SORTING and UNIQUE-ELEMENTS. When the "pictures" on the cards are just the integers in $[R]$, MEMORY can be thought of as a special case of SORTING where only a subset of $[R]^{2n}$ consisting of n equal pairs are valid inputs. An algorithm for SORTING can be used to output all n pairs in order of increasing variable values, effectively generating the output for MEMORY. In UNIQUE-ELEMENTS, the input consists of n integers $x_1, x_2, \ldots, x_n \in [n]$ and the desired output is a list of all i such that the value x_i appears exactly once in the input. Consider the following closely related problem that we call UNIQUE-PAIRS: given an input of $2n$ integers in $[n]$, output all pairs (i, j) with $i < j$ such that $x_i = x_j$ and no other $k \in [2n]$ satisfies $x_i = x_j = x_k$. Then, with minor modifications, the analysis that Beame gives for UNIQUE-ELEMENTS [Bea91] still applies, so this variant has $ST = \Omega(n^2)$. (See Appendix A for some details of how to modify Beame's proof.) So playing MEMORY is an easier task than both SORTING and UNIQUE-ELEMENTS, and thus it is harder to prove a lower bound. Indeed, so far we are only able to show $ST^2 = \Omega(n^3)$ for our problem, instead of the stronger $ST = \Omega(n^2)$ bound known for these related problems.

Our other lower bound for MEMORY, which does give an optimal tradeoff of $ST = \Omega(n^2 \log n)$, applies in a weaker model where cards may only be compared for equality. Its proof is based on a direct adversarial argument in the style of classic lower bounds for deterministic query complexity; see, e.g., Chapter 5 of the textbook by Du and Ko [DK00].

[2]The notation $[N]$ denotes the set $\{1, 2, \ldots, N\}$.

3 Preliminaries

Without loss of generality, we may assume that the $2n$ cards in a game of MEMORY are laid down on the table linearly. We model our input as a $(2n)$-tuple of variables $\mathbf{x} = (x_1, \dots, x_{2n})$, where x_i is the "picture" on the ith card. We think of each picture as an integer in $[R] := \{1, \dots, R\}$. We model the act of flipping the cards as reading the values of these variables. Per the rules of MEMORY, the only *valid* tuples $\mathbf{x}$ are those where exactly n distinct values occur exactly twice each. We let $\mathscr{X}$ denote the set of valid inputs. Given $\mathbf{x} \in \mathscr{X}$, a *match* in $\mathbf{x}$ is a triple (i, j, v) where $1 \leqslant i < j \leqslant 2n$, $1 \leqslant v \leqslant R$, and $x_i = x_j = v$. Clearly, every valid $\mathbf{x}$ has exactly n matches. The goal of MEMORY is to output all n matches—this is how we model the act of removing all cards—under the promise that the input is valid.

It is reasonable to assume that $R = n^{O(1)}$: indeed, if R were larger, we could simply hash each picture down to a $\lceil 3 \log n \rceil$-bit string using a 2-universal hash function. The idea of universal hashing is to select the hash function at random from a specific class of functions at the beginning of the execution. Therefore, no single input will always lead to worst case behavior. Formally, a family of hash functions $\mathscr{H}$ that map a finite universe U of keys into $[m]$ is *2-universal* if for each pair of distinct keys $k, l \in U$, the fraction of hash functions $h \in \mathscr{H}$ for which $h(k) = h(l)$ is at most $1/m$. Picking $U = [R]$, $m = O(n)$ would lead to a collision with probability at most $O(1/n)$. See Section 11.3.3 in the textbook by Cormen et al. [CLRS09] for more on universal hashing.

We are requiring the output to specify not just the matching pairs of cards (i, j) but also the integer values shown on those cards. This, too, is reasonable because it costs at most n additional variable reads to satisfy this requirement, and at least n variables must be read by any correct strategy.

For the rest of this thesis, we shall study computational complexity in worst-case settings for inputs. More precisely, an (S, T)-strategy is required to use at most S bits of memory. In the deterministic case, the strategy must always terminate with a correct output after reading at most T variables. In the randomized case, it must always terminate with a correct output and the expected number of variables it reads must be at most T. This is a Las Vegas notion of randomization. In contrast, a Monte Carlo algorithm may sometimes produce an incorrect solution (we usually try to bound the probability of correctness). See Chapter 1.2 in Motwani and Raghaven's textbook [MR95] for more on different notions of randomization.

We proceed to formally define two computational models for playing MEMORY using limited memory (space). We then present a basic, and easy to prove, upper bound that applies to the weaker of these models. We shall eventually prove a tight lower bound in this weaker model and a non-tight lower bound in the stronger model.

3.1 Computational Models

Branching Programs. Our strongest model is that of *general branching programs* on the variables $x_1, \dots, x_{2n}$. Recall that each variable takes values in $[R]$. An R-way branching program (BP) is a directed acyclic graph where each node is labeled with one of the variables x_i; there are exactly R out-edges from each non-sink node, labeled with the R different possible values for x_i; an edge is additionally annotated with zero or more outputs; and there is a single *source node*. Given an input $\mathbf{x}$, the execution of the BP on $\mathbf{x}$ starts at the source node. At each time step, the algorithm reads the variable x_i labeling the *current node* and, based on its value, it branches to one of the R successors of the node. In the process, it produces all the outputs that annotate the edge traversed. If the algorithm moves to a *sink node*, then it halts. See figure 6 for an example.

We can model an (S, T)-strategy for MEMORY as an R-way *layered* branching program $\mathscr{B}$. This is simply a branching program whose nodes are arranged in layers, numbered from 0 to T, with each edge going from one layer to the next. The source node is in layer 0 and every node in layer T is a sink. Further, since the strategy is limited to S bits of working memory (space), each layer has at most 2^S nodes. It is conventional

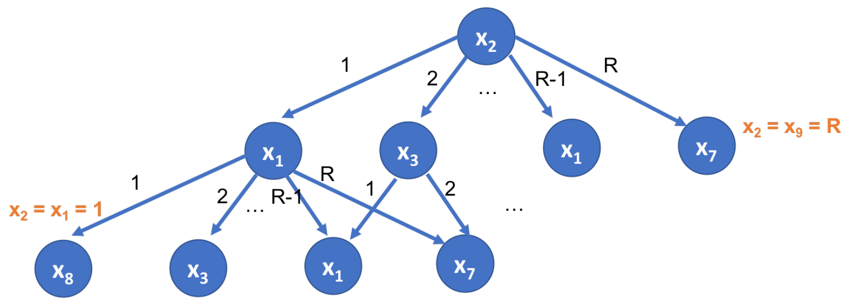


Figure 6: Example of an R-way branching program

to say that $\mathscr{B}$ has length T and width at most 2^S.

On every input $\mathbf{x} = (x_1, \ldots, x_{2n}) \in [R]^{2n}$, running $\mathscr{B}$ on $\mathbf{x}$ will trace a source-to-sink path through its nodes, outputting some triples in the process. Correctness requires that for every valid input $\mathbf{x} \in \mathscr{X}$, exactly n outputs are produced, namely, the n matches in $\mathbf{x}$.

The Blind Player Model. Since the two main actions in MEMORY are remembering some cards and comparing two cards for equality, the game can also be studied in a weaker model where such equality comparisons are the *only* thing one is allowed to do with the pictures on the cards. The BP model above allows a strategy to perform arbitrary computations (e.g., arithmetic) with the integers written on the cards. In contrast, in a *blind player model*, a strategy cannot see the pictures on the cards directly, but can only know whether or not two such pictures are equal. A strategy can, however, manipulate other relevant integers, such as the index positions at which particular cards appear.

More formally, an blind player (S,T)-strategy maintains a *working set* $\mathscr{S} \subseteq [2n]$: its elements are the indices of all cards "remembered" by the strategy. The set $\mathscr{S}$ is initially empty, and such that $|\mathscr{S}| \log n \leqslant S$ at all times, so that $\mathscr{S}$ fits in S bits of memory. In each time step, the strategy *examines* a card. Suppose it examines the ith card: then an oracle instantaneously informs it whether $x_i = x_j$, for each $j \in \mathscr{S}$. Based on this information, the strategy outputs zero or more values and updates $\mathscr{S}$ to a subset of $\mathscr{S} \cup \{i\}$.

3.2 A Basic Upper Bound

We proceed to analyze a natural strategy, implementable in the blind player model, that sets the benchmark upper bound $ST = O(n^2 \log n)$.

Proposition 1. *In the blind player model described above, for all S with $\log n \leqslant S \leqslant n \log n$,* MEMORY *has an (S,T)-strategy where $ST = O(n^2 \log n)$.*

Proof. Let $s := \lfloor S/\log n \rfloor$ be the number of cards that will fit in memory. The algorithm proceeds in *passes* and each pass begins by clearing the working set to $\varnothing$. In pass i, the player reads $x_{(i-1)s+1}, \ldots, x_{is}$ and stores them all in memory, outputting any matches found. The player then reads $x_{is+1}, \ldots, x_{2n}$ in order, forgetting each card after it is read, and outputting any matches found between these cards and the ones stored in memory. Clearly, every match in the input is eventually found and each pass takes $O(n)$ time.

This algorithm makes at most $\lceil 2n/s \rceil$ passes, for a total time complexity of $T = O((2n/s) \cdot n)$. This gives $ST = O(n^2 \log n)$, as desired. □

Note that our model does not account for the space used by a strategy to keep track of its progress, e.g., through loop indices. This is just to keep things simple: had we accounted for this, the space usage of the above strategy would increase by only an additive $O(\log n)$.

4 Warm-up: A Lower Bound for a Blind Player

We shall now prove that the simple upper bound in Proposition 1 is tight in the blind player model. In fact, we can prove something stronger. Consider an (S,T)-strategy for MEMORY and let $s := \lfloor S/\log n \rfloor$ as above. Since each examination of a card reveals the answer to at most s queries of the form "$x_i = x_j$?", it suffices to prove that $\Omega(n^2)$ queries must be made by any *query strategy* that can access information about $\mathbf{x}$ only through such pairwise queries. It then follows that $T = \Omega(n^2/s)$, giving $ST = \Omega(n^2 \log n)$.

To analyze such a query strategy, consider the *knowledge graph* G of the algorithm, defined as follows: G is an undirected graph with vertices $1,2,\ldots,2n$ (the indices of the cards); edge $\{i,j\}$ occurs in G iff $x_i = x_j$ is consistent with the validity of the input $\mathbf{x}$ and all answers to queries received by the algorithm. At the beginning of the game, G is a complete graph (figure 7). At all times, every edge of G is *useful*, where a *useless edge* is defined to be one that does not belong to any perfect matching (figure 8). Further, as the algorithm proceeds, G can only lose edges. The algorithm can conclude that $x_i = x_j$ iff $\{i,j\}$ is an isolated edge in G (figure 9). It is done when and only when G becomes a perfect matching (figure 10).

We shall also allow the algorithm the following information for free: we can only have $x_i = x_j$ if exactly one of i and j is in $[n]$. This means that, at the start of the algorithm, the knowledge graph G is a complete bipartite graph with "left side" $L = [n]$ and "right side" $R = [2n] \setminus [n]$ (figure 11).

Theorem 2. *Every query strategy for* MEMORY *using pairwise equality queries must make $\Omega(n^2)$ queries in total. Therefore, an (S,T)-strategy in the blind player model requires $ST = \Omega(n^2 \log n)$.*

Proof. We shall prove our lower bound by an adversarial argument, using the following adversary. Every time the query algorithm asks whether $x_i = x_j$, the adversary answers "No" unless consistency requires a "Yes" answer. This means that the knowledge graph G evolves as follows, with each query made.

- The algorithm asks "Is $x_i = x_j$?" for some non-isolated edge $\{i,j\}$ in G. The adversary answers "No", causing this edge to be *deleted* from G.
- The algorithm considers each remaining edge in G and checks whether it is still useful. The edges which are not useful are said to *vanish* from G.

When the algorithm finishes, G must reduce to a perfect matching: call it M. Further, exactly $n(n-1)$ edges must have been either deleted or vanished from G. Each query results in exactly one deletion, so it suffices to prove that $\Omega(n^2)$ edges get deleted by the time G becomes M. The trouble is that a deletion *could* be accompanied by up to $\Omega(n^2)$ vanishings: for instance, consider the deletion of the edge $\{n,n+1\}$ from the bipartite graph whose edge set is $\{\{n,n+1\}\} \cup \{\{i,n+j\} : 1 \leqslant i \leqslant j \leqslant n\}$ (figure 12).

The key is to observe that the edges outside M can be partitioned into pairs so that, within each pair, the first of the edges to be removed from G must be deleted, rather than vanished (figure 13). To formalize this, we first renumber the vertices so that final matching M is $\{\{i,n+i\} : 1 \leqslant i \leqslant n\}$. Now, for each $e \notin M$ in the initial complete bipartite graph, define $\phi(e)$ as follows:

$$\text{If } e = \{i,n+j\}, \text{ then } \phi(e) := \{j,n+i\}\,.$$

On the edges outside M, this mapping ϕ is an involution with no fixed points, so it partitions those edges into pairs, as required.

Claim. For all $e \notin M$, either e or $\phi(e)$ was deleted from G, and not vanished.
Proof of Claim. Suppose not. Then, either e and $\phi(e)$ were vanished as a result of the same query, or they were vanished as a result of two different queries. In the former case, consider the state of G immediately after the deletion of the edge involved in this query. At this point G contained both e and $\phi(e)$, and so both

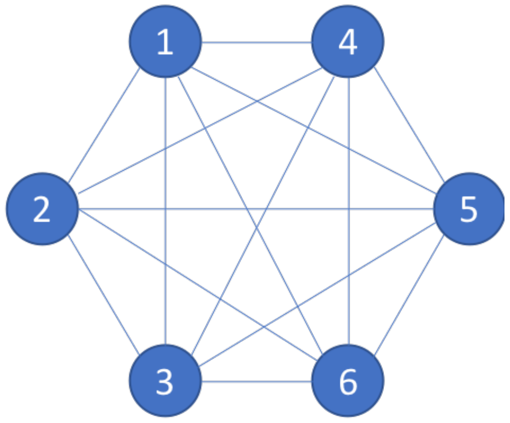


Figure 7: Starting configuration of the knowledge graph when $n = 3$

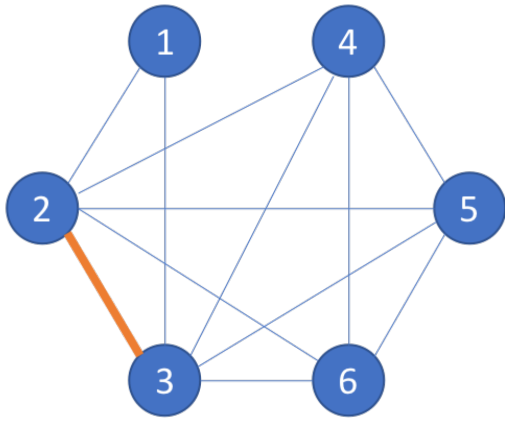


Figure 8: Edge (2,3) is useless, so it vanishes

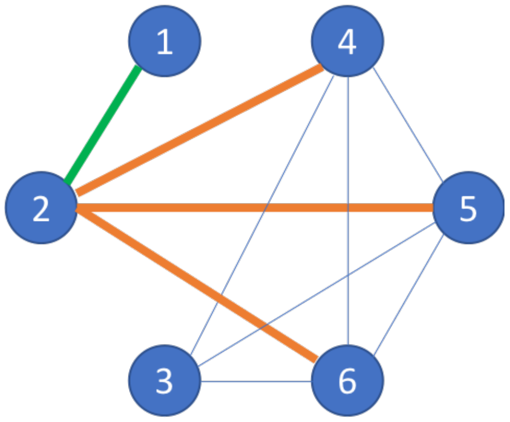


Figure 9: (2,4), (2,5),(2,6) vanish; (1,2) is matched

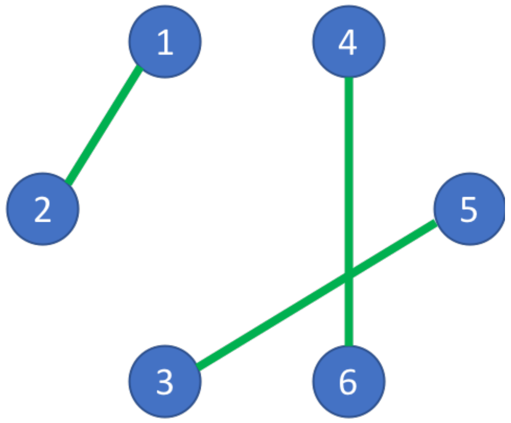


Figure 10: G becomes a perfect matching

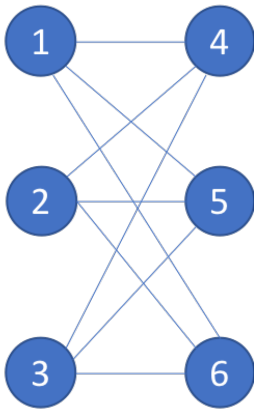


Figure 11: Starting configuration is a bipartite graph

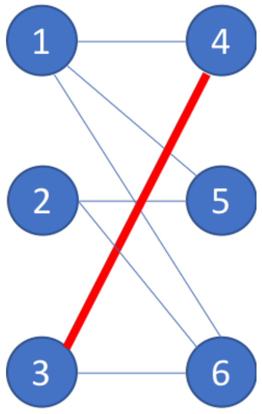


Figure 12: Deleting (3,4) can cause (2,6),(1,6),(1,5) to vanish

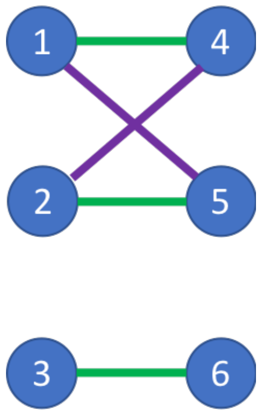


Figure 13: Suppose (1,4), (2,5), (3,6) are the final matching. Then either (1,5) or (2,4) has to be deleted.

these edges belonged to the perfect matching $(M \setminus \{\{i, n+i\}, \{j, n+j\}\}) \cup \{e, \phi(e)\}$. Therefore both these edges were useful and should not have vanished, a contradiction.

In the latter case, suppose (without loss of generality) that e was the first of the two edges to vanish. Consider the state of G immediately after the deletion of the edge involved in the query that caused e to vanish. Arguing as above, e is useful at this point and so it should not have vanished, a contradiction. This proves the claim.

Therefore, the total number of edges deleted from G is at least $n(n-1)/2 = \Omega(n^2)$, as required. □

5 The Main Lower Bound

5.1 The Overall Setup

Let $\mathscr{B}$ be a layered branching program of length T and width at most 2^S that correctly plays the Memory Game. Let r and t be positive-integer-valued parameters to be chosen later. We divide $\mathscr{B}$ into *stages*, where each stage, except perhaps the last, consists of $r+1$ consecutive layers of nodes of $\mathscr{B}$, and the final layer of nodes of stage i equals the first layer of nodes of stage $i+1$ (figure 14). Call a stage *productive* for an input

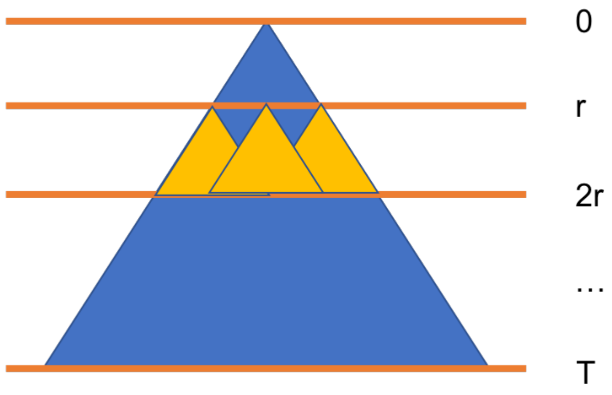


Figure 14: Divide a branching program into stages of depth r

$\mathbf{x} \in \mathscr{X}$ if running $\mathscr{B}$ on $\mathbf{x}$ produces at least $2t$ outputs in that stage. Recall, from Section 3, that $\mathscr{X}$ denotes the set of *valid* inputs.

Assume, towards a contradiction, that the number of stages is at most $n/(2t)$. Then, for every input $\mathbf{x}$, there exists a stage productive for $\mathbf{x}$. Therefore, there exists a stage i productive for a set $\mathscr{X}'$ of inputs, where $|\mathscr{X}'|/|\mathscr{X}| \geqslant 2t/n$. Consider the first layer of nodes in this stage. There are at most 2^S such nodes, so one of these nodes—v, say—is reached by at least a 2^{-S} fraction of the inputs in $\mathscr{X}'$.

Consider the subprogram of $\mathscr{B}$ with source node v, consisting of only the nodes in stage i. Unfold this subprogram into a decision tree with output (by replicating nodes as needed) and, if necessary, increase its depth to r by querying some dummy variables (figure 15). Let $\mathscr{T}$ be the resulting tree. Then $\mathscr{T}$ produces at least $2t$ correct outputs for a $(2t/n)2^{-S}$ fraction of inputs in $\mathscr{X}$.

Set $r = \lfloor (2/e)\sqrt{nt} \rfloor$. The main technical argument in our proof shows that a tree with this depth is "too shallow" and is therefore quite unproductive. Specifically, by Lemma 5, which we prove below, the fraction of inputs in $\mathscr{X}$ for which $\mathscr{T}$ correctly produces at least $2t$ outputs is at most $e^{-t} + (n/2)^{-t}$. Therefore,

$$\frac{2t}{n} \cdot 2^{-S} \leqslant e^{-t} + \left(\frac{n}{2}\right)^{-t} .$$

Setting $t = S$ gives us a contraction.

Therefore, for the above choices of r and t, the branching program $\mathscr{B}$ must have more than $n/(2t)$ stages. Since each non-last stage has length r, it follows that

$$T \geqslant \frac{nr}{2t} = \frac{n\lfloor (2/e)\sqrt{nS} \rfloor}{2S} = \frac{\Omega(n^{3/2})}{\sqrt{S}} ,$$

which proves the tradeoff $T\sqrt{S} = \Omega(n^{3/2})$. This outlines proves our main result.

Theorem 3 (Main Theorem)**.** *Any deterministic branching program of length T and width at most 2^S that correctly plays the Memory Game must obey the tradeoff $ST^2 = \Omega(n^3)$.*

In the rest of Section 5, we fill in the necessary details to formally prove Theorem 3. In the sequel, we indicate how to generalize the lower bound to randomized strategies.

5.2 A Probability-Theoretic Lemma

Our eventual proof of the unproductivity of shallow decision trees crucially hinges on a probability-theoretic lemma that we now develop.

Suppose we have a bag of $2n$ balls, two of each color. We pick r balls without replacement from the bag. Let Y be the number of pairs we picked (each pair has the same color). We want to bound the probability that $Y \geqslant t$. Formally, suppose r elements are chosen, without replacement, from the set $[n] \times \{0,1\}$, forming a random subset A. Define the random variable

$$Y = |\{j \in [n] : (j,0) \in A \text{ and } (j,1) \in A\}| .$$

Note that $Y = \sum_{j=1}^{n} I_j$, where $I_j = \mathbb{1}_{(j,0)\in A \wedge (j,1)\in A}$. These indicator random variables $I_1, \ldots, I_n$ are clearly not independent. Nevertheless, we can prove the following strong tail estimate on Y.

Lemma 4. *For all $t \geqslant 1$,*

$$\Pr[Y \geqslant t] \leqslant \exp\left(-t \ln \frac{4nt}{er^2}\right) .$$

In particular, for $r \leqslant (2/e)\sqrt{nt}$, we have $\Pr[Y \geqslant t] \leqslant e^{-t}$.

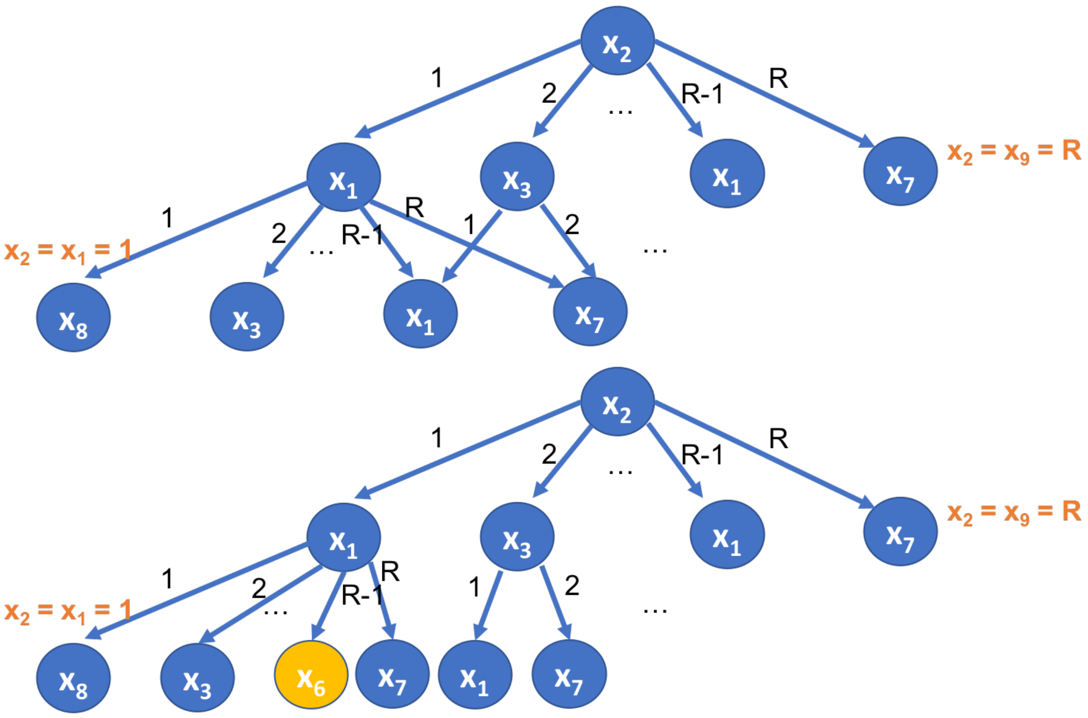


Figure 15: Unfold a subprogram into a decision tree. Subtrees rooted at x_1, x_7 in layer 2 are replicated. x_1 in layer 2 is turned into a dummy variable x_6 so that no path makes redundant queries.

Proof. For $1 \leqslant j \leqslant n$, put $U_j = \mathbb{1}_{(j,0)\in A}$ and $V_j = \mathbb{1}_{(j,1)\in A}$, so that $I_j = U_j V_j$. We shall use some terminology and results from Joag-Dev and Proschan [JP83]. The tuple $(U_1, V_1, \ldots, U_n, V_n)$ has a permutation distribution [JP83, Definition 2.10]. Therefore, these random variables are negatively associated [JP83, Theorem 2.11]. Since the variables I_j are increasing functions of disjoint subsets of these variables, they too are negatively associated [JP83, Property P_6]. Invoking a result from Dubhashi and Panconesi [DP09, Theorem 3.1], we can “apply Chernoff-Hoeffding bounds as is” to the sum $Y = \sum_{j=1}^{n} I_j$.

To be precise, note that

$$\mathbb{E}Y = \sum_{j=1}^{n} \mathbb{E}I_j = n \cdot \frac{r}{2n} \cdot \frac{r-1}{2n-1} \leqslant \frac{r^2}{4n}\,.$$

So consider the binomially distributed random variable $Z \sim \mathrm{Bin}(n, p)$, where $p = r^2/(4n^2)$. Then, for all $t > 0$, we have $\Pr[Y \geqslant t] \leqslant \Pr[Z \geqslant t]$. We shall use the following very precise Chernoff bound; see, e.g., Theorem 1 of Arratia and Gordon [AG89]. For all reals a, p, with $0 < p < a < 1$,

$$\Pr[Z \geqslant an] \leqslant e^{-n\mathrm{D}(a\|p)}\,, \tag{3}$$

$$\text{where } \mathrm{D}(a\|p) = a \ln \frac{a}{p} + (1-a)\ln\frac{1-a}{1-p} \tag{4}$$

is the relative entropy of the Bernoulli distribution $\mathrm{Bern}(a)$ to $\mathrm{Bern}(p)$. We simplify (4) as follows.

$$\mathrm{D}(a\|p) \geqslant a\ln(a/p) + (1-a)\ln(1-a) \geqslant a\ln(a/p) - a = a\ln(a/(ep))\,.$$

Using this in (3), with the setting $a = t/n$, gives us the claimed bound on $\Pr[Y \geqslant t]$. □

5.3 The Unproductivity of Shallow Decision Trees

We now come to the main technical thread of the proof. As shown in Section 5.1, proving this lemma will complete the proof of the tradeoff lower bound $ST^2 = \Omega(n^3)$.

Lemma 5 (Main technical lemma)**.** *Consider a subprogram of $\mathscr{B}$ of depth $r \leqslant \lfloor n/2 \rfloor$. Then for every $t \leqslant \lfloor r/2 \rfloor$, the probability that a uniformly-random input would produce at least $2t$ outputs following this subprogram is at most $(n-r-t)^{-t} + e^{-t}$.*

Proof. Unfold this subprogram into a decision tree with output (by replicating nodes as needed). If necessary, remove any redundant nodes in the tree, where an already-queried variable is re-queried, and increase the depth of every leaf to r by querying some dummy variables. Let $\mathscr{T}$ be the resulting tree. Then, in $\mathscr{T}$, along every path from the root to a leaf, exactly r inputs are queried; no input is queried and no output is generated more than once along any path.

Let Π be the set of all root-to-leaf paths in $\mathscr{T}$. Let $\mathbf{x}$ be a uniformly random input. Let $\pi(\mathbf{x})$ denote the path that $\mathbf{x}$ follows in $\mathscr{T}$.

For each path $\pi \in \Pi$, let $s(\pi)$ denote the number of outputs along π. Suppose those $s(\pi)$ outputs are (y_k^1, y_k^2, v_k), for $1 \leqslant k \leqslant s(\pi)$. Recall that each such output is a declaration that $x_{y_k^1} = x_{y_k^2} = v_k$. Let $q_1(\pi), q_2(\pi), \ldots, q_r(\pi)$ be the indices of the positions queried along π, in the order that they are queried: thus $q_i(\pi) \in [2n]$ for each $i \in [r]$. Suppose the results of the queries are $x_{q_i(\pi)} = w_i(\pi)$ for each $i \in [r]$. Define an ordered list $W(\pi) = [w_1(\pi), w_2(\pi), \ldots, w_r(\pi)]$. Then there is a bijection between such an ordered list and a path; i.e. no two distinct paths π_1, π_2 have $W(\pi_1) = W(\pi_2)$ and any ordered list of r numbers picked from $\{1,1,2,2,3,3,\ldots,R,R\}$ corresponds to a path.

Let $p(\pi) = \Pr[\pi(\mathbf{x}) = \pi]$ denote the probability that a uniformly random input follows path π. Then

$$p(\pi) = \frac{|\{\mathbf{x} \in \mathscr{X} : x_{q_i(\pi)} = w_i(\pi), \forall i : 1 \leqslant i \leqslant r\}|}{|\mathscr{X}|} = \frac{|\{\mathbf{x} \in \mathscr{X} : x_i = w_i(\pi), \forall i : 1 \leqslant i \leqslant r\}|}{|\mathscr{X}|},$$

since all indices are symmetric.

For a uniformly-random input $\mathbf{x}$, let random variable X be the total number of equal pairs in $W(\pi(\mathbf{x}))$.

Let Y be defined as in Lemma 4, above. Clearly, Y is the number of equal pairs found upon querying $x_1, x_2, \ldots, x_r$ for a uniformly random input. We now argue that $X \equiv Y$ in distribution, i.e., that $\Pr[X = u] = \Pr[Y = u]$, for all u with $0 \leqslant u \leqslant \lfloor r/2 \rfloor$. Let $\Pi_u \subseteq \Pi$ denote the set of all paths in $\mathscr{T}$ with exactly u equal pairs in their r queries. Then

$$\Pr[X = u] = \Pr[\pi(\mathbf{x}) \in \Pi_u] = \sum_{\pi \in \Pi_u} p(\pi) = \frac{\sum_{\pi \in \Pi_u} |\{\mathbf{x} \in \mathscr{X} : x_i = w_i(\pi), \forall i : 1 \leqslant i \leqslant r\}|}{|\mathscr{X}|} = \Pr[Y = u].$$

Since $X \equiv Y$, it follows from Lemma 4 that

$$\Pr[X \geqslant t] = \Pr[Y \geqslant t] \leqslant e^{-t}.$$

Call a path π in $\mathscr{T}$ *good* if and only if it produces at least $2t$ outputs and at least t of those outputs are of two variables which have both been queried along π. Call an input $\mathbf{x}$ *good* if and only if $\pi(\mathbf{x})$ is good. So the probability that a uniformly-random input is good satisfies

$$\Pr[\pi(\mathbf{x}) \text{ is good}] \leqslant \Pr[X \geqslant t] \leqslant e^{-t}.$$

When an input passes through a bad path π, either π does not generate enough outputs, or it generates at least $t+1$ outputs of two variables that the path does not both query. Pick $t+1$ such outputs arbitrarily.

Among those $t+1$ output pairs, suppose k pairs have exactly one variable queried along π, and $p = t+1-k$ pairs have neither variables queried along π.

We first consider the k outputs with exactly one variable queried. Denote those k queried variables as $x_{b_1}, x_{b_2}, \ldots, x_{b_k}$. In total, there are at least $n-r$ variables that π does not query. For a uniformly random input consistent with queries along π, all of those unqueried variables are equally likely to match with x_{b_1}. Therefore, the probability that the variable matched with x_{b_1} is correct is at most $1/(n-r)$. For $1 \leqslant i \leqslant k-1$, suppose the variables matched with $x_{b_1}, \ldots, x_{b_i}$ are correct, we have at least $n-r-i$ equally likely choices to match with $x_{b_{i+1}}$. Therefore, the probability that $x_{b_1}, x_{b_2}, \ldots, x_{b_k}$ are all matched correctly is at most

$$\left(\frac{1}{n-r}\right)\left(\frac{1}{n-r-1}\right)\cdots\left(\frac{1}{n-r-k+1}\right) \leqslant (n-r-k+1)^{-k}.$$

Next we consider the p outputs with neither variable queried. Denote the *values* of those p pairs as $v_{c_1}, v_{c_2}, \ldots, v_{c_p}$. In total, there are at least $R-r$ possible values of variables that π does not discover in its queries. For a uniformly random input consistent with queries along π, all of those values of variables are equally likely to be the value of any of the p pairs. Therefore, the probability that $v_{c_1}, v_{c_2}, \ldots, v_{c_p}$ are all correct values is at most

$$\left(\frac{1}{R-r}\right)\left(\frac{1}{R-r-1}\right)\cdots\left(\frac{1}{R-r-p+1}\right) \leqslant (n-r-p+1)^{-p}.$$

Therefore, if we uniformly randomly draw one input $\mathbf{x}$ from the set of all bad inputs, the probability that $\mathbf{x}$ generates at least $2t$ correct outputs along $\pi(\mathbf{x})$ satisfies

$$\begin{aligned}\Pr[\mathbf{x} \text{ produces at least } 2t \text{ correct outputs} \mid \mathbf{x} \text{ is bad}] &\leqslant \Pr[\text{Guesses of } k+p \text{ matches are all correct}]\\ &\leqslant (n-r-k+1)^{-k}(n-r-p+1)^{-p}\\ &\leqslant (n-r-t)^{-t-1}\\ &< (n-r-t)^{-t}.\end{aligned}$$

Therefore, the total probability of a uniformly random input $\mathbf{x}$ (either good or bad) producing at least $2t$ correct outputs satisfies

$$\begin{aligned}\Pr[\mathbf{x} \text{ produces at least } 2t \text{ correct outputs}] &< (n-r-t)^{-t}\Pr[\mathbf{x} \text{ is bad}] + \Pr[\mathbf{x} \text{ is good}]\\ &< (n-r-t)^{-t} + e^{-t},\end{aligned}$$

which completes the proof. □

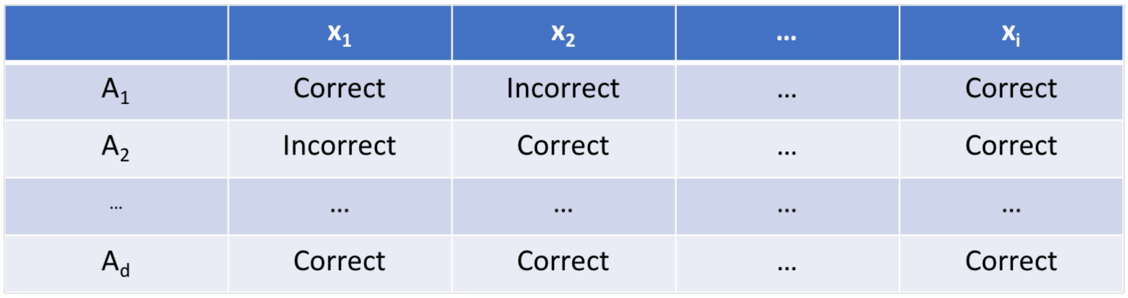
5.4 Lower Bound for Randomized Algorithms

The lower bound proved so far applies only to *deterministic* strategies for the Memory game. However, the ideas readily extend to give a similar lower bound for randomized strategies. We sketch this generalization.

Given space constraint S, suppose a Las Vegas algorithm $\mathscr{A}$ for the Memory game takes expected time $T = T(n,S)$. Remember that a Las Vegas algorithm always produces the correct output, but the run time varies from run to run. Convert $\mathscr{A}$ to a Monte Carlo algorithm $\mathscr{A}'$ that exactly mimics $\mathscr{A}$ expect that it always stops after $10T$ steps, at which point it outputs "Error". By Markov's inequality, for any input,

$$\Pr[\mathscr{A}' \text{ is incorrect}] = \Pr[\mathscr{A} \text{ has run time greater than } 10T] \leqslant \frac{T}{10T} = \frac{1}{10}.$$

Therefore, $\mathscr{A}'$ is correct for any input with probability $9/10$.



	x_1	x_2	...	x_i
A_1	Correct	Incorrect	...	Correct
A_2	Incorrect	Correct	...	Correct
...	...	...	...	...
A_d	Correct	Correct	...	Correct

Figure 16: Table demonstrating Yao's minimax principle

A randomized algorithm is simply a distribution over deterministic algorithms. Suppose $\mathscr{A}'$ is a distribution over deterministic algorithms $A_1, A_2, \ldots, A_d$. For all $j \in [d]$, A_j runs in time $10T$. Suppose $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_i$ are all the possible inputs in $\mathscr{X}$. We can make a table where row j and column k shows whether A_j is correct on input $\mathbf{x}_k$ (figure 16).

Yao's minimax principle [Yao77] says that if $\mathscr{A}'$ is correct 9/10 of the time in any column, then there exists a row that is correct on 9/10 of the columns. In other words, there is a deterministic algorithm A_j running in time $10T$ that generates n correct outputs for a set $\mathscr{X}''$ of inputs, where $|\mathscr{X}''|/|\mathscr{X}| \geqslant 9/10$. Model A_j as a branching program $\mathscr{B}$ with length $10T$ and width at most 2^S.

Similar to the proof for the deterministic lower bound, we divide $\mathscr{B}$ into stages of length $r = \lfloor (2/e)\sqrt{nS} \rfloor$. Assume that the number of stages is at most $n/(2S)$. Then there exists a subtree at some stage that produces at least $2S$ correct outputs for a $(2S/n)2^{-S}$ fraction of inputs in $\mathscr{X}''$. We arrive at the contradiction that

$$\frac{9}{10} \cdot \frac{2S}{n} \cdot 2^{-S} \leqslant e^{-S} + \left(\frac{n}{2}\right)^{-S}.$$

So $\mathscr{B}$ has more than $n/(2S)$ stages. Since each non-last stage has length r, it follows that

$$10T \geqslant \frac{nr}{2S} = \frac{n\lfloor (2/e)\sqrt{nS} \rfloor}{2S} = \frac{\Omega(n^{3/2})}{\sqrt{S}},$$

which proves the tradeoff $T\sqrt{S} = \Omega(n^{3/2})$.

6 Concluding Remarks

In this work, we studied the complexity of the Memory Game (also known as "Concentration") in a limited-memory setting, proving a nontrivial time-space lower bound. We showed that, when the player has S bits of memory, the number of card flips required to guarantee completion of the game is $\Omega(n^{3/2}/\sqrt{S})$, for both deterministic and randomized algorithms.

Our results suggest a number of directions for future work.

First, we conjecture that every randomized query strategy for MEMORY using pairwise equality queries must make $\widetilde{\Omega}(n^2)$ queries in total, i.e., that our Theorem 2 extends to the randomized case.

Second, it is worth investigating whether there is any (randomized) algorithm that can achieve $ST^2 = O(n^3)$. Beame et al. [BCM13] gives the same upper bound for ELEMENT-DISTINCTNESS as our lower bound for MEMORY under a strong notion of randomization. The core technique of their algorithm is Floyd's cycle-finding algorithm (also known as "the tortoise and the hare algorithm"), which finds a colliding pair with small space in $O(\sqrt{n})$ time. It is questionable whether this technique is strong enough to obtain the same upper bound for MEMORY, as MEMORY requires finding not just 1 but all n pairs.

Third, we could hope to obtain a stronger bound in a comparison branching program model. In this more restricted model, each node in the branching program compares two variables x_i, x_j and branches three ways, corresponding to the three cases $x_i < x_j$, $x_i = x_j$, and $x_i > x_j$, respectively. We conjecture that a stronger bound $ST = \Omega(n^{2-\varepsilon})$ is achievable in this model. Specifically, Borodin et al. [BFadH$^+$87] obtains a time-space tradeoff similar to ours for ELEMENT-DISTINCTNESS (the problem of outputting 1 if n input variables $x_1, \ldots, x_n$ are all distinct and 0 otherwise) in the comparison branching program model. Yao [Yao94] improves their bound to $ST = \Omega(n^{2-\varepsilon})$ through enforcing the space bound more frequently. We conjecture that an analogous improvement can also be applied to MEMORY, where the notion of progress may be the number of comparisons made between either identical pairs or cards with adjacent values when the n distinct integers on the $2n$ cards are sorted into increasing order.

Fourth, it is worth investigating what sort of tradeoffs can be obtained in a linear branching program model, where each node v performs linear queries $\sum_{i=1}^{n} \lambda_i^v x_i : c^v$ and branches three ways, corresponding to the three cases $\sum_{i=1}^{n} \lambda_i^v x_i < c^v$, $\sum_{i=1}^{n} \lambda_i^v x_i = c^v$, and $\sum_{i=1}^{n} \lambda_i^v x_i > c^v$, respectively. Yao [Yao82] proves that any branching program using linear queries to sort n inputs $x_1, x_2, \ldots, x_n$ must satisfy $ST = \Omega(n^2)$. This is done via a reduction to branching program using queries "$min(R) = ?$" where R is any subset of $\{x_1, \ldots, x_n\}$. Similarly, Karchmer [Kar86] proves a $ST^2 = \Omega(n^3)$ lower bound for ELEMENT-DISTINCTNESS in the linear branching program model. It may be possible to reduce from linear-query to min-query branching programs for MEMORY in a similar way. The analysis would also rely on a notion of progress which is perhaps the number of high-ranking comparisons made between either identical pairs or cards with adjacent values.

This work was motivated in part by the general question of what time-space tradeoffs one can obtain for the problem (in fact, family of problems) that calls for finding a large matching in an input graph. There are a number of interesting space-bounded algorithms for approximate maximum matching in the data streaming model; see, e.g., Crouch and Stubbs [CS14] and the references therein. There are also some corresponding lower bounds for this problem in a one-pass streaming model; see, e.g., Goel et al. [GKK12] and Assadi et al. [AKL17]. The streaming model is a very restrictive model for space-bounded algorithms: it requires the input to be accessed in a sequential fashion. One-pass streaming is even more restrictive. Yet, the precise tradeoff between space and approximation quality for maximum matching remains open even in the one-pass case, and there are essentially no nontrivial lower bounds in the multi-pass case. We hope that the lessons learned in studying the Memory game will find applications in establishing time-space trade-offs for finding large matchings in graphs. One can think of the Memory game as a graph on $2n$ vertices where only equal pairs are connected by an edge. What we have shown here are tradeoffs for discovering this perfect matching that is promised to exist.

Acknowledgments

We would like to thank Graham Cormode, Peter Winkler, and Paul Beame for helpful discussions about the problems considered in this work.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[Abr87] Karl Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, December 1987.

[Abr90] Karl Abrahamson. A time-space tradeoff for boolean matrix multiplication. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, FOCS '90, pages 412–419, vol. 1, Oct 1990.

[AG89] R. Arratia and L. Gordon. Tutorial on large deviations for the binomial distribution. *Bulletin of Mathematical Biology*, 51(1):125–131, 1989.

[AKL17] Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1723–1742, 2017.

[Alf07] Erik Alfthan. Optimal strategy in the children's game memory. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, May 2007.

[BC82] Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982. Preliminary version in *Proc. 12th Annual ACM Symposium on the Theory of Computing*, pages 294–301, 1980.

[BCM13] Paul Beame, Raphael Clifford, and Widad Machmouchi. Element distinctness, frequency moments, and sliding windows. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS '13, pages 290–299, Washington, DC, USA, 2013. IEEE Computer Society.

[Bea91] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. Preliminary version in *Proc. 21st Annual ACM Symposium on the Theory of Computing*, pages 197–203, 1989.

[Bea08] Paul Beame. Lecture notes on time-space tradeoff lower bounds using branching program, May 2008.

[BFadH$^+$87] A. Borodin, F. Fich, F. Meyer auf der Heide, E. Upfal, and A. Wigderson. A time-space tradeoff for element distinctness. *SIAM Journal on Computing*, 16(1):97–99, 1987.

[BO83] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 80–86, New York, NY, USA, 1983. ACM.

[BS90] Ravi B. Boppana and Michael Sipser. Handbook of theoretical computer science (vol. a). chapter The Complexity of Finite Functions, pages 757–804. MIT Press, Cambridge, MA, USA, 1990.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CS14] Michael Crouch and Daniel S. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, pages 96–104, 2014.

[DK00] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity*. John Wiley and Sons, New York, NY, USA, 2000.

[DP09] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, New York, NY, USA, 2009.

[FW13] Klaus-Tycho Foerster and Roger Wattenhofer. The solitaire memory game. Technical report, ETH Zurich, Switzerland, 2013.

[GKK12] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[JP83] Kumar Joag-Dev and Frank Proschan. Negative association of random variables, with applications. *Ann. Stat.*, 11(1):286–295, 1983.

[Kar86] Mauricio Karchmer. Two time-space tradeoffs for element distinctness. *Theoretical Computer Science*, 47:237 – 246, 1986.

[MNT93] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theoretical Computer Science*, 107(1):121–133, 1993.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[Sav97] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[VW13] Daniel J. Velleman and Gregory S. Warrington. What to expect in a game of memory. *The American Mathematical Monthly*, 120(9):787–805, 2013.

[Yao77] A. C. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 222–227, Oct 1977.

[Yao82] Andrew Chi-Chih Yao. On the time-space tradeoff for sorting with linear queries. *Theoretical Computer Science*, 19(2):203 – 218, 1982.

[Yao94] Andrew Chi-Chih Yao. Near-optimal time-space tradeoff for element distinctness. *SIAM Journal on Computing*, 23(5):966–975, 1994.

[Yes84] Yaacov Yesha. Time-space tradeoffs for matrix multiplication and the discrete fourier transform on any general sequential random-access computer. *Journal of Computer and System Sciences*, 29(2):183–197, 1984.

A A Lower Bound for the Unique Pairs Problem

In section 2.2, we compared the hardness of MEMORY to UNIQUE-ELEMENTS. Here we show that a time-space tradeoff of $TS = \Omega(n^2)$ can be obtained for a variant of MEMORY that is a harder task (and thus easier to prove a lower bound).

Consider the following problem of UNIQUE-PAIRS: Given an input of $2n$ integers in $[n]$, output all pairs (i, j) with $i < j$ such that $x_i = x_j$ and no other $k \in [2n]$ satisfies $x_i = x_j = x_k$. Modeling the proof for UNIQUE-ELEMENTS in [Bea08], we now prove that any algorithm computing UNIQUE-PAIRS that runs in time at most T and uses space at most S has $TS = \Omega(n^2)$.

Proof. For a uniformly random input over $[n]^{2n}$, the expected output size for UNIQUE-PAIRS is

$$\mathbb{E}\left[\#\text{ outputs}\right] = \binom{n}{2}\frac{1}{n}\left(1-\frac{1}{n}\right)^{2n-2} > \frac{n-1}{2e^2}$$

By Markov's inequality, the probability that the output size is at least $(n-1)/(4e^2)$ is

$$\Pr\left[\#\text{ outputs} > \frac{n-1}{4e^2}\right] \geqslant c$$

for some constant $0 < c < 1$.

Let $\mathscr{B}$ be the layered branching program computing UNIQUE-PAIRS. We have $T \geqslant 2n$. For some h to be chosen later, we partition $\mathscr{B}$ into T/h stages of depth h. We call an input *good* if UNIQUE-PAIRS generates at least $(n-1)/(4e^2)$ outputs for that input. So $\mathscr{B}$ run on a good input should produce at least

$$m = \frac{n-1}{4e^2(T/h)} = \frac{(n-1)h}{4e^2 T}$$

outputs in some stage. Same as the argument in [Bea08], for $m \leqslant n/4$ and $h \leqslant n/4$, for any $\mathscr{B}$ of depth at most h,

$$\Pr[\mathscr{B}\text{ produces at least } m \text{ correct outputs}] \leqslant e^{-c'm}$$

for some constant $c' > 0$. Choosing $h = n/4$, we have $m \leqslant n/4$. Since the fraction of good inputs is at least c, we need

$$2^S e^{-c'm} \geqslant c$$

which gives us $S = \Omega(m) = \Omega(n^2/T)$. □