The Computational Complexity Column

BY

Michal Koucký

Computer Science Institute, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
koucky@iuuk.mff.cuni.cz
https://iuuk.mff.cuni.cz/~koucky/

Simulating Fast Algorithms With Less Memory*

Ryan Williams
MIT and Institute for Advanced Study
rrw@mit.edu

Abstract

My goal is to provide a friendly background exposition of the recent theorem in computational complexity that $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[\sqrt{t \log t}]$. In English, the theorem says that for every problem that can be solved by a multitape Turing machine running in time t, there is another machine which solves the same problem using only $O(\sqrt{t \log t})$ space. The reader will determine whether or not this goal was met. ☺

1 Introduction

Many basic efficient algorithms consume an amount of memory that's proportional to their running time. For intricate algorithms where brand new information is constantly being computed from past results, it is natural to allocate a little bit of new space in every step or every other step, to store this new information for future use. For example, a dynamic programming solution, in which one builds a table with complex dependencies on previously-computed results, may require a table with its number of cells proportional to the total number of steps taken by the algorithm. In other words, any *particular* algorithm A with $t(n)$ running time may well require $\Theta(t(n))$ units of space. Is this level of space required by *all* algorithms computing the same function as A?

> **Question:** *Are there problems solvable in $t(n)$ time which require $\Omega(t(n))$ units of space to solve?*

*This material is based upon work supported by the National Science Foundation under grants DMS-2424441 (at IAS) and CCF-2420092 (at MIT).

To get a handle on the question, let's view it a little differently. Take any algorithm A that runs in time $t(n)$ (and which may use $\Theta(t(n))$ space). Does there always exist another algorithm B which is equivalent in functionality to A, but uses significantly less space than $O(t(n))$? Observe that a no-answer to this question is equivalent to a yes-answer to our original **Question**.

There does not seem to be any intuitive reason to believe that such an algorithm B always exists, for every algorithm A. It is easy to imagine that some algorithm A on n-bit input x can perform a sequence of operations that are so intricate and sophisticated, that along the way A produces some $t(n)$-bit string Y_x which is *pseudorandom*, or *unpredictable*, or *incompressible*, in some quantifiable sense. Why would (or should) there be an algorithm B which on input x can effectively produce all the bits of Y_x as needed, but uses drastically less memory than $\Omega(t(n))$ bits? It would only take one "weird" A capable of producing nastily incompressible strings on its inputs, in order for our **Question** to have a yes-answer.

A surprising answer. Amazingly, a sequence of three papers starting with Hopcroft, Paul, and Valiant in 1975 [8, 14, 7] showed that the answer to the **Question** is **NO** for essentially all reasonable models of computation.[1] Namely, these papers show that $t(n)$-time algorithms can always be simulated in only $O(t(n)/\log t(n))$ space![2] In complexity notation, we have the inclusion of classes

$$\mathsf{TIME}[t(n)] \subseteq \mathsf{SPACE}[t(n)/\log t(n)],$$

and this inclusion holds for a variety of underlying computational models.

However, there is a practical caveat to these simulation results: while the resulting algorithm B (simulating time-$t(n)$ algorithm A) runs in $O(t(n)/\log t(n))$ space as desired, B also takes running time $2^{O(t(n)^\varepsilon)}$ for $\varepsilon > 0$ (but ε can be as small as desired). Therefore, we've exchanged a tiny-looking log-factor improvement in space usage, for an exponential blow-up in the running time. Hopcroft, Paul, and Valiant [8] also show that a time-space tradeoff is possible: for all reasonable functions $b(n)$, the (multitape Turing machine) algorithm B can be implemented in $O(t(n)/\log\log b(n))$ space and $O(b(n) \cdot t(n))$ time. So, for example, there is a B running in $O(t(n)/\log\log t(n))$ space and $t(n) \cdot 2^{(\log t(n))^\varepsilon}$ time, for every $\varepsilon > 0$.

Why should we care? What is the use of such a simulation, showing that time-t computations can be simulated in $o(t)$ space but using substantially more time?

[1] Note that these three papers were titled "On Time Versus Space", "On Time Versus Space II", and "On Time Versus Space III", respectively. It took a surprising level of self-control for me to resist naming my contribution "On Time Versus Space IV".

[2] We also note that the paper [8] cites an earlier work by Paterson-Valiant [16] which showed that every Boolean circuit of size $s(n) \geq n$ has an equivalent circuit of depth $O(s(n)/\log s(n))$, an equally surprising result in circuit complexity.

On the one hand, if one allows the simulation time to be exponential, such a simulation would become useless in practice. On the other hand, if one uses the polynomial-time version of the simulation instead, a log-log-factor improvement in space would apparently not be too noticeable. One hypothetical application would be to increase the computational ability of a device which has low memory and little access to the rest of us, such as an aging deep space probe. We imagine this probe has significantly less on-board memory than a modern computer, but has particular inputs (observations it can make) which are unavailable to computers on Earth. Instead of trying to send those inputs back to Earth, which could be intractable for various reasons, the probe could apply a $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[o(t)]$ type result to use a more memory-restricted version of a time-efficient algorithm, allowing strictly more computations to be performed in principle. Despite this colorful scenario, it is still not obvious (to me) that there is a concrete practical application of $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[o(t)]$ results when the time complexity of the $o(t)$-space simulation is large.

Nevertheless, there are at least two good non-practical (theoretical) answers to the question of why we should care.

1. Time and space are fundamental resources for computing, and understanding how the two relate to each other is one of the most basic problems that we could study. Any non-trivial result relating the TIME and SPACE complexity classes is worth knowing.

2. We can use such surprising simulations to prove impossibility results (a.k.a. lower bounds) for simulating space-bounded computations quickly. The famous P vs PSPACE question is equivalent to asking whether $\mathsf{SPACE}[n]$ is contained in $\mathsf{TIME}[n^k]$ for some constant $k \geq 1$. Using the simulations of [8, 14, 7], we can conclude that $\mathsf{SPACE}[n]$ is not contained in $\mathsf{TIME}[o(n \log n)]$. Otherwise, we would be able to derive that

$$\mathsf{SPACE}[n^2 \log n] \subseteq \mathsf{TIME}[o(n^2 \log^2 n)] \subseteq \mathsf{SPACE}[o(n^2 \log n)],$$

   a contradiction to the Space Hierarchy Theorem [19]. Thus these simulations provide a very modest time lower bound against linear space.

Should we expect a better simulation? While simulations of TIME in smaller SPACE are indeed useful for proving lower bounds, the approach seems like overkill. To prove $\mathsf{SPACE}[n] \not\subseteq \mathsf{TIME}[T(n)]$ for *large* $T(n)$, we only have to find *some* hard problem in linear space that cannot be solved in $T(n)$ time. This seems to be much easier (and more likely to be true) than simulating *every* problem in $\mathsf{TIME}[T(n)]$ inside of $\mathsf{SPACE}[o(n)]$. In the words of Ben Brubaker of Quanta magazine [2], the approach "feels almost cartoonishly excessive, akin to proving

a suspected murderer guilty by establishing an ironclad alibi for everyone else on the planet."

Apparently it was believed that the $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[t/\log t]$ simulation could not be improved by much. First of all, there were tight lower bounds for the general approach: the various strategies of all prior work [8, 14, 7] can be captured by a simple pebble game on directed acyclic graphs, and extensive work [15, 10] showed that there are graphs on which all the time-space tradeoffs proved are optimal. Thus, it was well-known that if the simulation could be improved, it would have to be done using very different techniques from prior work.

At some point, researchers began exploring the consequences of assuming that Hopcroft-Paul-Valiant and its successors cannot be improved in a substantial way. In one of the first derandomization papers giving serious evidence that $\mathsf{P} = \mathsf{RP}$, Sipser [20] showed that assuming a certain expander conjecture (which was later proven to be true [21]) and assuming the conjecture that $\mathsf{TIME}[t] \not\subseteq \mathsf{SPACE}[t^{1-\varepsilon}]$ for all $\varepsilon > 0$, we can conclude $\mathsf{P} = \mathsf{RP}$. The later pioneering work by Nisan and Wigderson [11] showed that assuming $\mathsf{TIME}[t] \not\subseteq \mathsf{SPACE}[t^{1-\varepsilon}]$ for all $\varepsilon > 0$ also implies a derandomization of BPP. Crucially, their work also shows the assumption $\mathsf{TIME}[t] \not\subseteq \mathsf{SPACE}[t^{1-\varepsilon}]$ can be relaxed to the assumption that $\mathsf{TIME}[t]$ does not have Boolean circuits of size $t^{1-\varepsilon}$ (i.e., $\mathsf{TIME}[t] \not\subseteq \mathsf{SIZE}[t^{1-\varepsilon}]$). That is, Nisan and Wigderson showed we do not need *space lower bounds against time* to obtain derandomization; we only need *circuit size lower bounds against time*. Our recent work simulating time in smaller space contradicts the conjecture that $\mathsf{TIME}[t] \not\subseteq \mathsf{SPACE}[t^{1-\varepsilon}]$ for all $\varepsilon > 0$, but it says nothing about the more plausible conjecture that $\mathsf{TIME}[t] \not\subseteq \mathsf{SIZE}[t^{1-\varepsilon}]$ for all $\varepsilon > 0$.

The new surprise. If you read the abstract of this paper, there is unfortunately no real surprise. ☺ The simulation of $\mathsf{TIME}[t]$ inside of $\mathsf{SPACE}[t/\log t]$ can be radically improved for the multitape Turing machine model:

Theorem 1.1 ([22])**.** *Every multitape Turing machine running in time $t(n)$ can be simulated by another multitape Turing machine using space only $O(\sqrt{t(n)\log t(n)})$.*

To appreciate the scope of this result, we note the surprising power of multitape Turing machines. Recall that these are simply Turing machines endowed with a constant number of tape heads that may independently access any constant number of tapes. (In each step of such a machine, each tape head reads its current cell, writes to its current cell, and may move left or right on the tape.) Already two-tape Turing machines appear to be quite powerful: the P-complete Circuit Evaluation problem [13], sorting [17], fast matrix multiplications, fast Fourier transforms (and integer multiplication), and many other basic algorithms [18] can be implemented on such devices with running times that are comparable (up to

polylogarithmic factors) to the best known running times on arbitrary random-access models of computation. In fact, it is a major open question to find any decision problem that can be solved in $O(T(n))$ time on a random access model of computation (for some $T(n) \geq n$), which cannot be solved in $T(n) \cdot \text{poly}(\log T(n))$ time on a two-tape Turing machine. Despite their restricted sequential access to tapes, multitape Turing machines can perform surprisingly quick *amortized* computations if they are allowed (at least linear) time to shuffle data around.

2 How Does It Work?

The main idea behind proving this result is to read the correct pair of papers at the same time, and to believe very strongly that the ideas of these papers can be successfully combined. The two papers to read are:

1. Hopcroft, Paul, and Valiant's "On Time Versus Space" from FOCS 1975 [8], and

2. Cook and Mertz's "Tree Evaluation in $O(\log n \cdot \log\log n)$ Space" from STOC 2024 [4]

We'll go through both of these soon. Prior to discussing either of them, we will introduce a graph-theoretic notion for reasoning about a time-t computation. This notion makes sense for essentially any reasonable serial model of computation, and it models the "flow of information" in the computation.

Given a "machine" M of some type and an input x to run on M, we define a *computation graph* $G_{M,x}$ in the following way.

First, there are $t + 1$ nodes labeled $0, 1, \ldots, t$, representing the steps of the computation. Each node i will be tagged with some bits of information info(i), which informally is the information computed by M in the i-th step. (For now, we think of info(0) as encoding the initial state of the machine, which is passed to step 1.) For example, if M is a multitape Turing machine, info(i) may simply denote the transition computed in the i-th step: the states at the beginning and end of the i-th step, the symbols read, the symbols written, and the head positions moved.

Second, we put a directed edge (i, j) in $G_{M,x}$ if and only if *info*(i) *is needed to compute info*(j). For example, in basically every computational model, we have the edges $(i - 1, i)$ for all $i \in [t] := \{1, \ldots, t\}$: we always need the "program counter" or "state" at the end of step $i - 1$ in order to compute step i. We also need an edge (i, j) if there is a "register" or "cell" written to in step i which is read next in step j. For reasonable models, we only access $O(1)$ memory locations in any one step, so the indegree of each node of $G_{M,x}$ is $O(1)$. Observe that such a directed graph is always acyclic.

The graph $G_{N,x}$ is *defined* so that the following important property holds. For every $j \in [t]$, let in-nbrs(j) be the set of all i such that (i, j) is an edge. Then we have the principle:

> *Given info(i) for all $i \in$ in-nbrs(i), we can compute info(j) in $O(1)$ steps.*

(Indeed, the computation we are doing is effectively simulating M on x for one step.) Our goal is to determine info(t), which will tell us whether the computation accepts or rejects.

What we need from HPV. Motivated by the above principle, HPV define a *pebble game* to be played on DAGs such as $G_{M,x}$. The game has the following rules:

1. We can always put a pebble on a source node.
2. Given pebbles on all nodes in in-nbrs(i), we can put a pebble on node i.
3. We can remove pebbles at any time.

Here, the act of placing a pebble directly corresponds to simulating one step of M on x. Therefore, a very interesting question is: how many pebbles do we need to pebble the nodes of an n-node graph? HPV show that for directed acyclic graphs of indegree $O(1)$, there is a pebbling algorithm which uses only $O(n/\log n)$ pebbles. This directly corresponds to a memory-bounded algorithm: in principle, we only need to keep around $O(t/\log t)$ "intermediate results" of the computation, in order to determine the final outcome. This strongly suggests that we only need $O(t/\log t)$ space to simulate a time-t computation.

However, we don't know this graph $G_{M,x}$ in advance, and we can't naively store this $(t + 1)$-node DAG in only $O(t/\log t)$ space. For multitape Turing machines, HPV get around this issue by showing how to *compress* the computation graph, by partitioning the time and space of the computation into pieces. (Later work extending their simulation to random-access models [14, 7] works by performing clever space-saving simulations which effectively model good pebbling strategies, *without* storing a computation graph explicitly. This point will be discussed further in the last section.) This compressed graph notion will prove very useful for us.

Fix a multitape Turing machine M and input x. For a parameter $b \in [1, t]$ (which we will think of here as close to $\sqrt{t}$), each tape of M (on x) is partitioned into *blocks* of b contiguous cells each, and the time of M (on x) is partitioned into *intervals* of b consecutive steps. Our compressed computation graph $G'_{M,x}$ has only $n := O(t/b)$ nodes, one for each interval. For an interval i, we define info(i)

to be all the information computed by M (on x) during interval i. This includes the state and tape head positions at the end of the interval, along with the contents of tape blocks accessed during interval i, at the end of the interval. We define info(0) to simply be the *initial configuration* of M on x: the initial state of M and initial head positions, along with the input x. Similarly to the definition of the original graph $G_{M,x}$, we put an edge (i, j) exactly when we need some data from info(i) to compute info(j).[3]

As each tape block has b cells and intervals are b steps long, at most two tape blocks may be accessed in an interval. Since there are a constant number of tapes, the indegree of each node in $G'_{M,x}$ is still $O(1)$. Also observe that, when we are given info(i) for all in-neighbors of a node j, we can compute info(j) in $O(b)$ steps.

We can view the problem of computing info(n) in the $(n + 1)$-node graph $G'_{M,x}$ as a special Circuit Evaluation problem, where we have a circuit of n nodes with wires given by the edges of the graph, and where every gate of the circuit takes in $O(b)$ bits of information from a constant number of in-neighbors, runs for $O(b)$ steps on this information, and outputs $O(b)$ bits of information.

As before with the graph $G_{M,x}$, we cannot really know the graph $G'_{M,x}$ in advance; it needs to be computed somehow.[4] Since our graph $G'_{M,x}$ has only $O(t/b)$ nodes, which will be approximately $\sqrt{t}$, we could enumerate over all possible such graphs in $O(t/b \cdot \log(t/b))$ space (in fact, due to their structure coming from multitape Turing machines, these graphs can be encoded in $O(t/b)$ bits, by guessing the head movements across tape blocks; see the paper [22]). For each candidate graph G'', we may attempt to simulate M on x by pebbling G''. If G'' is not suitable, then it must "violate" the head movements of M on x in some way: such a violation will arise if we are missing an edge (i, j) but we needed info(i) in order to compute info(j). If we find that some edge is missing for us, then we move on to the next possible G''. We know that the $O(t/b)$-node $G'_{M,x}$ can be pebbled using only $O(t/b)/\log(t/b)$ pebbles, and we know that storing each pebble (which is just info(i) for various i) takes $O(b)$ space. Therefore for $b = t^{\varepsilon}$ for some $\varepsilon \in (0, 1)$, the overall space usage is $O(t/\log t)$.

What we need from Cook and Mertz. Cook and Mertz [4] study an apparently unrelated problem, introduced by (another) Cook et al. [5], called Tree Evaluation. For the purposes of this article, we will think of Tree Evaluation as the following problem. We are given:

[3] Hopcroft-Paul-Valiant also add other technical conditions, such as making the machine M "block-respecting", but they aren't really necessary.

[4] If M was an "oblivious" Turing machine with structure, then we could compute the edges of $G_{M,x}$, but making an arbitrary multitape Turing machine oblivious requires extra time overhead. See the paper [22] for details.

- a rooted tree T of height h, where each node of T has d children and $d \leq O(1)$,
- a function $f : (\{0,1\}^b)^d \to \{0,1\}^b$, presented as a table of $2^{d \cdot b}$ values, each of which are b-bits long, and
- for each leaf ℓ of T, we are given a value $v_\ell \in \{0,1\}^b$.

Note that the tree T could be very large, and have nearly d^{h+1} nodes in total. For each inner node u of T with children $u_1, \ldots, u_d$, we inductively define the value v_u to be $f(v_{u_1}, \ldots, v_{u_d})$ (we concatenate the values $v_{u_1}, \ldots, v_{u_d}$ and feed the db-bit result as input to f). Starting from the leaves, this yields a value v_u for every u in T. The goal of Tree Evaluation is to compute v_r, where r is the root of T.

We can apply the pebble game to any tree T, which we can think of as a directed acyclic graph where the children have arcs to their parents. A simple depth-first pebbling strategy shows that every Tree Evaluation instance (construed as a DAG) can be pebbled using $O(h)$ pebbles, storing at most two pebbles at each level of the tree. Since each node value takes b bits to describe, this corresponds to an evaluation algorithm using $O(h \cdot b)$ space. For pebbling, [5] show that we can't do better.

Incredibly, Cook and Mertz [4] show how to use only $O(h \log b + b)$ space, replacing the *product* of h and b with nearly the *sum* of h and b. In trying to describe Cook and Mertz's algorithm to others, Arthur C. Clarke's quote often comes to my mind: *Any sufficiently advanced technology is indistinguishable from magic* [3]. We are very accustomed to thinking about the space complexity of algorithms in particular rigid ways: viewing the space usage in terms of some stack, bounding the maximum height of the stack, bounding the "width" a.k.a. the number of bits needed on one layer of the stack, and so on. For Tree Evaluation, our stack height is h and our width is b. Cook and Mertz's algorithm provides a genuinely new way of reusing space in a recursive algorithm, inspired by work in *catalytic* computation which was initiated by Buhrman et al. [1]. They still have a stack of height h, but its width is only $O(\log b)$. Instead of having b bits at every layer of the stack, there is a common storage of $O(b)$ bits which is reused many times over, at every layer of the stack. Time and space prevent us from describing their algorithm in detail; beyond their paper, see [6, 22] for alternative expositions of their great result.

What does this have to do with computation graphs? We have a surprisingly space-efficient algorithm for evaluating *trees* in which each leaf holds a b-bit value, and each inner node of the tree computes a function from $O(b)$ bits to b bits, from children to parent. The problem of evaluating $G'_{M,x}$ amounts to evaluating a *circuit* or *directed acyclic graph* in which each source holds a b-bit value,

and each non-source node computes a function from $O(b)$ bits to b bits, taking in values from its in-neighbors.

A folklore result in Boolean circuit complexity states that every Boolean circuit C of depth h (where each gate has arbitrary fan-out) can be computed by an equivalent Boolean *formula* F of depth h, where each gate has fan-out 1. The idea is, starting from the output gate of C, to "unroll" C backwards: we perform a depth-first traversal of C by traversing arcs backwards, in effect enumerating over all paths from the output gate to some source node of C. Doing so, we obtain a *formula* where each gate of the formula can be given a *name* by specifying a path from some gate of C to the output gate of C. The formula F may contain many copies of gates from C, but its overall depth will be the same as that of C. Moreover, this transformation is space-efficient: given the encoding of C and a path P from a gate g to the output of C, the path P *names* a gate in the formula F. We can easily compute the names of the children of P in the formula F, using the circuit C. For an n-gate circuit C of depth d and $O(1)$ fan-in, it takes only $O(d)$ space to specify a gate in F (by walking backwards from the output of C), and to compute the names of its children.

We can think of Boolean circuits as a special case of a computation graph where the block size $b = 1$, and we can think of Boolean formula evaluation as a special case of Tree Evaluation where $b = 1$. The above transformation from circuits to formulas works equally well for circuits and formulas which pass along b-bit values on all their wires, where $b > 1$. Therefore we can apply the Cook-Mertz procedure for Tree Evaluation to a candidate computation graph G'', in order to simulate a multitape Turing machine M on an input x. Setting $b = \sqrt{t}$ to be the length of a time interval, the total number of intervals becomes $\Theta(t/b) = h = \Theta(\sqrt{t})$ and the simulation runs in $O(\sqrt{t} \cdot \log t)$ space overall. To minimize the space usage, we want to set

$$b = h \log(b).$$

Slightly adjusting b to be $\sqrt{t \log t}$, making the intervals a little longer, the number of intervals becomes $O(\sqrt{t/\log t})$ and we obtain the final $O(\sqrt{t \log t})$ space bound.

3 What's Next?

I believe that currently the most significant open problem is to extend the simulation of $\mathsf{TIME}[t]$ in $\mathsf{SPACE}[\sqrt{t \log t}]$ to more general random-access models of computation (RAMs), or to give compelling evidence that this cannot be done. A weaker but still very interesting result would be to show that time-$t(n)$ on RAMs can be simulated in space $O(t(n)^{1-\varepsilon})$ for *some* $\varepsilon > 0$. Let me outline some ideas for how this latter problem might be solved.

Concretely, let us suppose we have a Turing machine with access to a *tree storage* [14]: instead of a tape storage with cells arranged in a line, the Turing machine receives its input written on a complete binary tree of depth $O(\log n)$, where each node of the tree is now a cell. The tape head starts at the root of the tree. At each step, the Turing machine reads from the current cell, changes its state, overwrites the current cell, and moves its head to either the parent of the current cell, the left child of the current cell, or the right child of the current cell. For all reasonable random-access models that we know of, those that run in time $t(n)$ can be simulated on a tree storage in $t(n) \cdot \mathrm{poly}(\log t(n))$ time. Thus it suffices to extend the space-efficient simulation to a tree storage. It does not seem that we can easily embed the computation graph for such a Turing machine into a nice Tree Evaluation instance that we can succinctly store and manipulate. But perhaps if we can go "beyond" Tree Evaluation—doing something similar to what Tree Evaluation does in spirit, without actually storing an explicit graph—then perhaps we can obtain a better simulation. This is precisely how the extensions of Hopcroft-Paul-Valiant to random-access models work [14, 7]: these extended simulations do *not* store a computation graph explicitly. Instead, they simply present particular recursive strategies for saving space, and argue *in their analysis* that their simulation corresponds to "pebbling" a t-node computation graph representing the RAM computation, where every node corresponds to a single step.

Another possible direction (towards a better simulation for random-access models) would be to improve the best-known $O(\sqrt{t(n)})$ space bound for simulating a one-tape Turing machine running in time $O(t(n))$ [9, 12]. It is not hard to show that every $O(t(n))$-time Turing machine with a tree storage can be simulated by a standard one-tape Turing machine in about $O(t(n)^2)$ time, by simply sweeping through the entire contents of the tree storage during every simulated step. Therefore if we could simulate time $T(n)$ on *one-tape* Turing machines in $O(T(n)^{1/2-\varepsilon})$ for some $\varepsilon > 0$ (a slight improvement over the multitape case), then we would have a new space-efficient simulation of $O(t(n))$-time RAMs as well.

References

[1] Harry Buhrman, Richard Cleve, Michal Kouckỳ, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 857–866. ACM, 2014. doi:10.1145/2591796.2591874.

[2] Ben Brubaker. For algorithms, a little memory outweighs a lot of time. Quanta Magazine, May 2025. URL: https://www.quantamagazine.org/for-algorithms-a-little-memory-outweighs-...-20250521/.

[3] Arthur C. Clarke. *Profiles of the future: an inquiry into the limits of the possible*. Harper & Row, New York; London, rev. ed. edition, 1973.

[4] James Cook and Ian Mertz. Tree evaluation is in space O(log n · log log n). In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1268–1278. ACM, 2024. URL: https://doi.org/10.1145/3618260.3649664.

[5] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, 2012. URL: https://doi.org/10.1145/2077336.2077337.

[6] Oded Goldreich. On the Cook-Mertz Tree Evaluation procedure. *Electron. Colloquium Comput. Complex.*, TR24-109, 2024. URL: https://eccc.weizmann.ac.il/report/2024/109.

[7] Joseph Y. Halpern, Michael C. Loui, Albert R. Meyer, and Daniel Weise. On time versus space III. *Math. Syst. Theory*, 19(1):13–28, 1986. URL: https://doi.org/10.1007/BF01704903.

[8] John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space. *J. ACM*, 24(2):332–337, 1977. Conference version in FOCS'75. URL: https://doi.org/10.1145/322003.322015.

[9] John E. Hopcroft and Jeffrey D. Ullman. Relations between time and tape complexities. *J. ACM*, 15(3):414–427, 1968. URL: https://doi.org/10.1145/321466.321474.

[10] Thomas Lengauer and Robert Endre Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *J. ACM*, 29(4):1087–1130, 1982. doi:10.1145/322344.322354.

[11] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994. URL: https://doi.org/10.1016/S0022-0000(05)80043-1.

[12] Mike Paterson. Tape bounds for time-bounded Turing machines. *J. Comput. Syst. Sci.*, 6(2):116–124, 1972. URL: https://doi.org/10.1016/S0022-0000(72)80017-5.

[13] Nicholas Pippenger. Fast simulation of combinational logic networks by machines without random-access storage. In *Proceedings of the Fifteenth Annual Allerton Conference on Communication, Control and Computing*, pages 25–33, 1977.

[14] Wolfgang J. Paul and Rüdiger Reischuk. On time versus space II. *J. Comput. Syst. Sci.*, 22(3):312–327, 1981. URL: https://doi.org/10.1016/0022-0000(81)90035-0.

[15] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Math. Syst. Theory*, 10:239–251, 1977. doi:10.1007/BF01683275.

[16] Mike Paterson and Leslie G. Valiant. Circuit size is nonlinear in depth. *Theor. Comput. Sci.*, 2(3):397–400, 1976. URL: https://doi.org/10.1016/0304-3975(76)90090-6.

[17] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *J. ACM*, 25(1):136–145, 1978. doi:10.1145/322047.322060.

[18] Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter. *Fast algorithms: a multitape Turing machine implementation*. Bibliographisches Institut, Mannheim, 1994.

[19] Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis II. Hierarchies of memory limited computations. In *6th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 179–190. IEEE Computer Society, 1965. URL: https://doi.org/10.1109/FOCS.1965.11.

[20] Michael Sipser. Expanders, randomness, or time versus space. *J. Comput. Syst. Sci.*, 36(3):379–383, 1988. URL: https://doi.org/10.1016/0022-0000(88)90035-9.

[21] Michael E. Saks, Aravind Srinivasan, and Shiyu Zhou. Explicit or-dispersers with polylogarithmic degree. *J. ACM*, 45(1):123–154, 1998. URL: https://doi.org/10.1145/273865.273915.

[22] R. Ryan Williams. Simulating time with square-root space. In Michal Koucký and Nikhil Bansal, editors, *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025*, pages 13–23. ACM, 2025. doi:10.1145/3717823.3718225.