The tight deterministic time hierarchy [1]

Martin Fürer

Department of Mathematics
University of Tübingen
D-74 Tübingen, Fed. Rep. of Germany

Abstract Let k be a constant ≥ 2, and let us consider only deterministic k-tape Turing machines.

We assume $t_2(n) > n$ and t_2 is computable in time t_2. Then there is a language which is accepted in time t_2, but not accepted in any time t_1 with $t_1(n) = o(t_2(n))$.

Furthermore, we obtain a strong hierarchy (isomorphic to the rationals $\mathbb{Q}$) for languages accepted in fixed space and variable time.

1. Introduction

It is well known (see e.g. [5]) that the slightest increase in the asymptotic growth from the function s_1 to the well-behaved function s_2 allows more languages to be accepted (i.e. more problems to be solved) by deterministic Turing machines in space s_2 than in space s_1 [1]. Whereas, it is only known that more languages are accepted in the well-behaved time t_2 than in time t_1, if $t_1(n) \log t_1(n) = o(t_2(n))$ (diagonalization method [2] combined with fast tape reduction [3]). For nice functions t_1 which are bounded by an exponential function (i.e. t_1 does not grow too much), the factor $\log t_1(n)$ can be reduced to $(\log t_1(n))^\varepsilon$ [7] (for every $\varepsilon > 0$) by applying the padding methods of Ruby and P.C. Fischer [9].

A much tighter result was obtained by Paul [6, 7] for a fixed number of tapes. For this case, he has reduced the factor $\log t_1(n)$ to $\log^* t_1(n)$. ($\log^* n$ is 0 for $n=1$ and defined by $1 + \log^* \lfloor \log_2 n \rfloor$ for $n > 1$.) We will prove here that no factor at all is necessary. $t_1(n) = o(t_2(n))$ is enough for a fixed number $k \geq 2$ of tapes.

1) This work was supported by the British Science Research Council and by the Swiss National Foundation

2. Fast simulation

The major difficulty in getting a tight hierarchy is to show that there exists a universal k-tape Turing machine which can simulate any k-tape Turing machine for a predetermined number of steps without losing any time for counting the steps.

The obvious thing to try is to keep the step counter (containing a nonnegative integer p) and the code of the simulated program q always near a head of the simulating Turing machine. In the program q the instruction currently executed is marked, and the counter is decreased by one before each step. Furthermore p and q are pushed to follow the head movement. But pushing p after each simulation step costs a factor $\log p$.

Paul [6] has noticed that it is not necessary to push the whole counter all the time. Pushing only a number of length $\log \log p$ is sufficient to simulate and count $\log p$ steps. After having simulated $\log p$ steps, we don't mind when the head of the simulating Turing machine spends $O(\log p)$ steps to walk back and collect the bigger part of the counter (containing now $p - \log p$). We have at least two tapes. Therefore the counter can be moved fast.

This method can be iterated. Instead of moving a counter of length $\log \ldots \log p$ (i times log) after each simulation step, a small subcounter of length $\log \log \ldots \log p$ ($i+1$ times log) is separated and the big counter is collected after $\log \ldots \log p$ (i times log) steps. It is not hard to see that the time to simulate p steps is $O(p \log^* p)$. And diagonalization yields Paul's result [6]. (The present author has discovered this independently but later. This has initiated this work, but for a long time Paul's result seemed to be the best possible.)

The whole problem can be seen as a distribution problem. Goods are stored in the counter p. They have to be consumed one piece at the time at the then current head position, i.e. at many different places. The cost of transport is proportional to the length of the distance, except when the distance is extremely short (shorter than the length of the counter).

Paul's solution [6] makes sure that transports are only made when absolutely necessary, and always the amount of goods transported is chosen as big as possible with low cost (i.e. extra cost for transports of big counters over short distances is avoided). We would like to accept this solution as optimal. How can we do better? The answer is: with predistribution. We can set up a chain of wholesale businesses and small shops which keep the goods ready before the consumer (i.e. the head on the tape) comes across. Because our goods are just integers this concept will be realized by a new representation of integers.

3. A counter which can be changed everywhere locally

A number in radix representation (say decimal) can be decreased by one at the right end, where the least significant digit is located. In the average, the carry propagation (actually it is a "borrow", i.e. a negative carry) does not go far, so that only digits near the right end are changed. We introduce a new redundant representation of non-negative integers where every location (not only the right end) has the property that in the average only digits near that location have to be changed in order to decrease the number by one.

We represent the number A by

$$A = \sum_{h \in X} \sum_{j \in Y} a_{hj} B^h \quad a_{hj} \in \{0,\dots,B-1\}$$

where $X = \{0,\dots,H\}$ and
$Y = \{0,\dots,2^{H-h}-1\}$.
Hence the coefficient of every B^h is instead of a digit a sum of several digits.
We arrange the a_{hj}'s in the nodes of a binary tree:

- a_{H0} is stored in the root,
- $a_{h-1\ 2j}$ is stored in the left and $a_{h-1\ 2j+1}$ in the right son of a_{hj} ,
- hence, h is the height of the node containing a_{hj} .

We define $b_{2^h(2j+1)} = a_{hj}$ (see Figure 1).

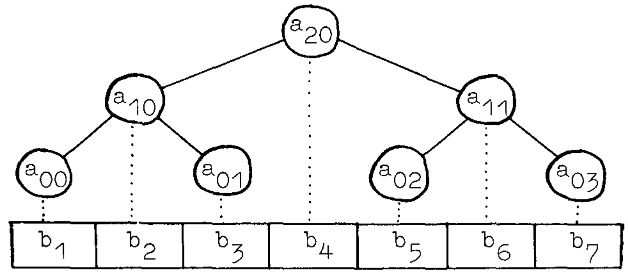

Figure 1: The tree of counters and its implementation on a one dimensional array.

Then at every odd position $2j+1$ we have a least significant digit b_{2j+1} .

Definition

The word $b_1 \dots b_{2^{H+1}-1}$ is called *a tree representation of height H* of the number

$$A = \sum_{h \in X} \sum_{j \in Y} b_{2^h(2j+1)} B^h ,$$

where $X = \{0,\dots,H\}$ and
$Y = \{0,\dots,2^{H-h}-1\}$
Carries are propagated from every son node to its father node, which is implemented nearby when these nodes are not high in the tree. And for $B > 2$ the average distance of the carry propagation is bounded by a small constant.

To see this, we note first that the distance from the position of a_{hj} to the position of its father is only 2^h in the array. Now let all a_{hj} be set to $B-1$ initially. Then we choose m times a $j \in \{0,\dots,2^H-1\}$ and subtract one from a_{0j} . Of course we propagate all the necessary carries as well. Hereby at most $\lfloor m/B \rfloor$ carries are necessary from nodes of height $h-1$ to nodes of height h. Hence for $B > 2$, the sum of the distances of carry propagation is at most

$$\sum_{h=1}^{H} \frac{m}{B^h} 2^{h-1} = \frac{m}{B} \frac{1-(2/B)^H}{1-(2/B)} < \frac{m}{B-2} = O(m) .$$

Therefore, when we start with full counters, then for every single run (i.e. sequence of subtractions by one), the average distance of carry propagation is less than one. This is the basic property of the tree representation.

Remark

We can also construct a tree representation of integers such that not only subtraction, but also addition of one (in any location of the tree representation) can be done with constant average carry propagation distance. We just allow the B-ary digits to be elements of $\{-(B-1),\ldots,0,\ldots,B-1\}$. But for the application in this paper, we don't need such a representation of integers.

4. Simulation in linear time

Let q be any reasonable code of a Turing machine M_q. This means, there is a constant c such that every step of M_q can be simulated in time $c|q|$, where $|q|$ is the length of the code q.

Lemma

For all $k \geq 2$, there is a k-tape Turing machine M which on any input of the form $w\#p\#q$ (where w is a word over the fixed alphabet Σ, p is a B-ary non-negative integer and q is the code of a k-tape Turing machine M_q over the alphabet Σ) simulates exactly p steps of M_q with input w. Furthermore M does this simulation in time $O(|w| + p|q|)$, i.e. in a time linear in p.

Outline of the proof

First we build a tree representation of a number p' with all counters full (i.e. B-1). We choose p' such that $p'' = p-p'$ is not negative and not much bigger than p'. We place this tree representation on a tape of M such that the head is in its center. If this tree representation of p' has length L, we will see that we can build it in time $O(L)$. The simulation determines the head movement. But the basic property of the tree representation implies that we can simulate $s \geq (L+1)/2$ steps without spending more than time $O(s)$ for counting the steps.

When the counter at the root is already 0, but we try to subtract one from it, then we set this counter to -1, and we interrupt the simulation. In time $O(L)$ we convert the tree representation to normal B-ary notation. Now we build a new tree representation with a smaller tree but with full counters.

When the head of the simulating Turing machine M leaves the area of the tree representation, then we also interrupt the simulation and move the tree representation to have the head again in the center. Alternatively we could do the same procedure as we do when the root counter gets -1 .

We present the details of the proof in the Appendix.

5. The tight time hierarchy

Definition

A function $t : \mathbb{N} \to \mathbb{N}$ is *time-constructible* on k-tape Turing machines, if there exists a deterministic k-tape Turing machine which reads the input n in unary notation and computes $t(n)$ in binary notation, while doing at most $t(n)$ steps.

Remark

This definition of Paul [7] is different from the definition of time-constructibility used in [5] and other places. The advantages of Paul's definition are:

1) For every well-behaved function t with $n = o(t(n)$ it is easy to see that t is time-constructible.
2) Time constructibility in this sense is what is actually used in hierarchy theorems.
3) Every function t which is fully time-constructible on k tapes in the traditional sense is also k-tape time-constructible in our sense. Actually a proof of 3) is easy to find, but it involves the methods developed in the preceding paragraph.

Definition

$DTIME_k(t)$ is the set of languages accepted by k-tape deterministic Turing machines in time t (i.e. for words of length n , the number of steps is at most $t(n)$) .

Well known diagonalization techniques (see e.g. [5]) together with the Lemma of the preceeding paragraph imply:

Theorem 1

If $k \geq 2$ and t_2 is time-constructible on k-tape Turing machines, then there is a language in $DTIME_k(t_2)$ which is not in $DTIME_k(t_1)$ for any t_1 with $t_1(n) = o(t_2(n))$.

Remark

This Theorem seems not to be true for d-dimensional tapes with $d > 1$. On the other hand, it is easy to see that Paul's result [6] (gap at most a factor $\log^* t_1(n)$) holds for any fixed number of tapes of fixed dimension d .

6. A time hierarchy for fixed space

In complexity theory, hierarchies obtained by diagonalization usually involve i.o. (infinitely often) lower bounds, i.e. the lower bounds hold (for every Turing machine accepting the language) for infinitely many inputs. And a.e. (almost everywhere) upper bounds correspond to these lower bounds. A slight modification makes it always possible to obtain a quantitative result about the density of difficult inputs. In particular, we can usually get lower bounds which hold for almost every input length.

When all lower bounds have to hold for almost every input length, then it is no more possible to transform a discrete hierarchy into a dense hierarchy by simply allowing more time only on certain subsets of the word lengths.

Paul [6] has proven an ω-hierarchy (there is a set of complexity classes ordered isomorphic to $\mathbb{N}$) for a fixed space-function and a sequence ot time-functions. This ω-hierarchy is a corollary to the following theorem.

Theorem 2 (Paul [6])

If the functions t and δ are both constructible on k tapes $(k \geq 2)$ in time t and $\log^* t(n) \leq \delta(n) \leq \log\log t(n)$, then $DTIME_k(t \cdot \delta) \subseteq DSPACETIME_k(t, t \cdot 2^{2^{\delta}})$. (Def. below).

By taking Paul's proof, but using the faster simulation, we obtain the same theorem without the condition $\log^* t(n) \leq \delta(n)$. This leads immediately to the conjecture, that we can obtain a dense hierarchy from the improved Theorem 2. This conjecture is correct, but it turned out that Paul's simulation also implies this dense hierarchy.

In order to get a strong hierarchy, where the lower bounds hold for almost all input lengths, we first strengthen Theorem 2.

Definition

$DSPACETIME_k(s, t)$ is the set of languages accepted by k-tape deterministic Turing machines simultaneously in space s and time t.

Definition

For $S \subseteq \mathbb{N}$ $\mathcal{L}(S)$ is the set of words in $\mathcal{L}$ whose lengths are in S.

Theorem 3

If the functions t and δ are both constructible on k tapes $(k \geq 2)$ in time t and $\delta(n) \leq t(n)$, then there is a language $\mathcal{L} \in DSPACETIME_k(t, t \cdot \delta)$, such that there is no language $\mathcal{L}'$ and no infinite set $S \subseteq \mathbb{N}$ with $\mathcal{L}(S) = \mathcal{L}'(S)$ and $\mathcal{L}' \in DTIME_k(t \cdot \log\log \delta)$.

Proof

It is easy to transform Paul's proof of Theorem 2 into a proof of Theorem 3. We replace δ by $\log\log\delta$, and instead of just doing a simulation, we also do a diagonalization. On input w, the Turing machine M_w is simulated with input w. To get the lower bound for almost every input length, we have to make sure that every Turing machine has codes of almost every length.

Theorem 4

For every (on $k \geq 2$ tapes) time-constructible function t with $t(n) \geq n+1$ (i.e. t is not bounded by a constant), there is a (densely ordered) set of functions $\{\delta_q : q \in \mathbb{Q} \cap (0,1)\}$ (η hierarchy), such that for every $q \in \mathbb{Q} \cap (0,1)$, there is a language $\mathcal{L}$ with $\mathcal{L} \in DSPACETIME_k(t, t \cdot \delta_q)$, but if S is infinite, $\mathcal{L}'(S) = \mathcal{L}(S)$ and $p < q$, then

$\mathcal{L}' \notin$ *DSPACETIME*$_k$ $(t, t\cdot\delta_p)$.

Proof

We have to solve the following problem: Find easy computable functions δ_q such that

(i) $\delta_q(n) = o(t(n))$ for all q and

(ii) $\delta_p(n) = o(\log\log\delta_q(n))$ for all $p > q$.

First we define functions σ_q by $\sigma_q(n) = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}\lfloor qn\rfloor$. For $n \geq \frac{3}{q-p}$, we have $qn \geq 3 + pn$, $\lfloor qn\rfloor \geq \lfloor 3+pn\rfloor$ and therefore $\sigma_p(n) < \log\log\log\sigma_q(n) = o(\log\log\sigma_q(n))$.

So the functions σ_q have the property (ii), but they grow too fast.

But the functions δ_q defined by $\delta_q(n) = \sigma_q(\log^* t(n))$ solve the problem.

Now by Theorem 3, there is a language $\mathcal{L} \in \mathrm{DSPACETIME}_k\ (t, t\cdot\delta_q)$ and for any infinite set $S \subseteq \mathbb{N}$ and any $\mathcal{L}'$ with $\mathcal{L}'(S) = \mathcal{L}(S)$ we have $\mathcal{L}' \notin \mathrm{DTIME}_k\ (t\cdot\log\log\delta_q)$. Therefore $\delta_p(n) = o(\log\log\delta_q(n))$ implies $\mathcal{L}' \notin \mathrm{DTIME}_k\ (t\cdot\delta_p(n))$ for all $p < q$. Hence $\mathcal{L}' \notin \mathrm{DSPACETIME}\ (t, t\delta_p)$.

7. A final remark

It is hoped that the tree representation of integers will have applications in other areas of computer science. Maybe in some connections a similar but a bit more complicated representation of integers might be useful. This representation was used in an earlier version of this paper. It still has the basic property of tree representation. The difference is that higher in the tree more digits are stored (they cannot be implemented successively on an array), and therefore the representation of a number p is much shorter, e.g. $\log_B^2 p$. In this paper the length of the representation of p is $L = p^{1/\log B}$.

Acknowledgment

Sven Skyum has helped me to find this elegant tree representation based on binary trees. This has simplified the proof of Theorem 1 significantly. Peter van Emde Boas has pointed out to me that an earlier version of Theorem 4 (where lower bounds have to hold only infinitely often) is a corollary to Paul's ω-hierarchy.

References

[1] Hartmanis, J., P.M. Lewis II, and R. E. Stearns, "Hierarchies of memory limited computations," Prof. Sixth Annual IEEE Symp. on Switching Circuit Theory and Logical Design (1965), pp. 179-190.

[2] Hartmanis, J., and R.E. Stearns, "On the computational complexity of algorithms," Trans. AMS 117 (1965), 285-306.

[3] Hennie, F.C., and R.E. Stearns, "Two-tape simulation of multitape Turing machines," J. ACM 13 1966, pp. 533-546.

[4] Hopcroft, J.E., W.J. Paul, and L.G. Valiant, "On time versus space," J. ACM 24 (1977), pp. 332-337.

[5] Hopcroft, J.E., and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, Mass. (1979).

[6] Paul, W.J., "On time hierarchies," Proc. Ninth Annual ACM Symposium on the Theory of Computing (1977), pp. 218-222.

[7] Paul, W.J., Komplexitätstheorie, Teubner Studienbücher Informatik, Stuttgart (1978).

[8] Paul, W.J., R.E. Tarjan, and J.R. Celoni, "Space bounds for a game on graphs," Mathematical System Theory 10 (1977), pp. 239-251.

[9] Ruby, S., and P.C. Fischer, "Translational methods and computational complexity," Proc. Sixth Annual IEEE Symp. on Switching Circuit Theory and Logical Design (1965), pp. 173-178.

Appendix:

Details of the linear time simulation

(Lemma, paragraph 4).

To describe the procedures for handling the tree representation of integers more precisely, we first define the storage organization.

tape 1

direct simulation of M_q	track 1
code q (current instruction marked)	track 2
tree representation of p'	track 3
R, L, T: right son, left son, top (= root)	track 4
current carry propagation path marked	track 5
rest p" = p-p'	track 6

Figure 2: The organization of tape 1.

Two work tapes, say tape 1 and tape 2 have six tracks each. Track 1 of each tape always contains a copy of the corresponding tape of the simulated Turing machine M_q. The purpose of the other tracks of tape 1 is described in Figure 2. Under each counter (= B-ary digit) in the tree representation of the positive integer p' , we write in track 4 , if the position of the counter in the tree is that of a left or right son or the top. The tracks 2 to 6 of tape 2 provide space to copy the corresponding tracks of tape 1. This copying is necessary to move information fast.

Now we describe the different procedures which have to be done from time to time during the simulation.

a) Construction of the tree representation

First we note that the value of a number in tree representation of height H with all counters full is

$$p' = \sum_{h=o}^{H} 2^{H-h}B^{h}(B-1) = \frac{B^{H+1}-2^{H+1}}{B-2}(B-1) < B^{H+2}$$

If p is less than a constant $c \geq B^2$ (typically c is chosen much bigger), then we don't construct a tree representation, but we do the counting during the simulation in normal B-ary notation. If $p \geq c$ then we choose $H = \lfloor \log_B p \rfloor - 2$. This implies $p' < p$ and the rest $p'' = p-p'$ is not too big. We compute p'' and store it in track 6.

Now the tree representation of p' can be put on track 3. It is just a sequence of $L = 2^{H+1}-1$ times the symbol $B-1$. To find the R (right), L (left) and T (top) markings for track 4, we note that the j-th marking $(j=1,\ldots,L)$ is R (resp. L, T) iff j in binary (without leading zeros) is of the form x110...0 (resp. x010...0, 10...0) .

It is clear that all this can be done in time $O(L)$.

b) Destruction of the tree representation

The values stored in the leaves (odd positions) are added to the rest p'' , which is first copied onto tape 2. Then all leaves are cut by copying only the even positions of the tree representation onto tape 2. Naturally the remaining tree is compressed to implementation length $\lfloor L/2 \rfloor$. This procedure is iterated, but each time the additions into p'' are done at the next higher B-ary position.

As for the construction, the time for destruction is only $O(L)$. In both cases, even $L^{\log B}$ would be fast enough, because at least $B^{H+1}-1 > L^{\log B}$ many simulation steps have been done in the meantime.

c) Carries

When we have to do a carry, then we mark its path on track 5. This path has either length 1 , or twice the length of the carry which has just been done. Doubling the path is easy, because we use a second tape. The markings on track 4 tell us on which side to find the father node. Before doing carries, we have put a distinguishing sign at the place where the simulation has to continue, and we always remember if this place is on the left or the right of the current head position. Then, finding back is easy.

The time is proportional to the length of the carry propagation path, which is shorter than 1 in the average (if no carry means length 0).

d) Shift of the tree representation

When the head of tape 1 leaves the area of the tree representation, then we move the tree representation by $(L+1)/2$. We can move the rest p'' with it, because the length of the B-ary representation of p'' is $O(L)$ (even $O(\log L)$). The time for the shift is $O(L)$ and is charged to the preceeding $(L+1)/2$ simulation steps.