On the notion of bit complexity

Claus Diem

August 21, 2010

Abstract

In many works in the fields of computational complexity, algorithmic number theory and mathematical cryptology as well as in related areas, claims on the running times of algorithms are made. However, often no computational model is given and the analysis is performed in a more or less ad hoc way, counting in an intuitive way "bit operations". On the other hand, the computational model of a successor RAM with logarithmic cost function provides an adequate and formal basis for the analysis of the complexity of algorithms from a "bit oriented" point of view.

This motivates the search for a result on the simulation of machines in a suitably defined general model by successor RAMs. In this work, a very general RAM model is defined, and then a "quasi-optimal" result on the simulation of such machines by successor RAMs is given.

1 Introduction

In a large body of works in the fields of computational complexity, algorithmic number theory and mathematical cryptography as well as in related areas, claims on the *running times* or *time complexity* of algorithms are made. However, in a substantial part of these works, the analysis of the algorithms is performed in a more or less intuitive and ad hoc way without reference to a specific model of computation.

Often, the running time (or expected running time) is computed by counting in some intuitive way "bit operations". Or to phrase it differently: in a certain intuitive way, the *bit complexity* of algorithms is considered. Such an approach is clearly sufficient if one is merely interested in questions on complexity from a "qualitative" point of view (disregarding exponents) – as is often the case in complexity theory. However, often more concrete statements are made, and then the question poses itself whether the claimed running time holds true in a particular "bit-oriented" model of computation.

The situation concerning *space requirements* or *space complexity* is similar but in fact – as we will discuss below – it is even worse because it is even less clear what is exactly meant by claims concerning space requirements.

This is by far a new phenomenon. Already in 1980 Arnold Schönhage observed; see [Sch80]:

> Many of the concrete algorithms given in the literature are (at least implicitly) designed for multitape Turing machines, sometimes the higher flexibility of random access machines (with a variety of instruction sets) is required, and frequently it is totally left to the reader's imagination what the model of computation should look like.

Let us fix the following terminology: By a *machine type* we mean a type of Turing machines, random access machines, etc.[1] A *model of computation* is then a machine type together with a time and a space measure. In some cases, these measures are obvious (e.g. for Turing machines), but in other cases – in particular for RAM models – they are not, and care has to be taken which measure is used.

It is intuitively obvious that if one speaks about running time without further comments, one should have a sequential machine type with a bit-oriented storage and atomistic instruction set in mind, and the time measure must reflect the number of bit operations required. Of course, we cannot give a rigorous definition of these intuitive notions but some requirements seem to be obvious: First, the machine type must have a most reduced set of instructions. Second, the time needed for one instruction must by definition reflect the lengths of the numbers which have to be considered for its execution.[2]

One such model is the multi-tape Turing machine model (with various similar definitions). Another model is the successor RAM with logarithmic cost measure (again with various similar definitions).

We note here that the bit-oriented point of view of this work is slightly different from the *atomistic* point of view in [Sch80]. An atomistic model according to Schönhage is for example the *Storage Modification Machine (SSM)*. However, from our point of view, the SSM model is not bit-oriented because it misses a bit-oriented (a priori) storage. Also, the successor RAM types (as defined in [Sch80]) with uniform time measure deserve to be called atomistic but not bit-oriented, because in one time unit arbitrarily many bits might be changed. Note here that it is shown in [Sch80] that the SSM

[1] We do not give a rigorous *general* definition of "machine type", and – consistent with this – we do not claim any *general* mathematical propositions on machine types. The mathematical propositions are rather for *specific* machine types.

[2] We assume here implicitly that the machines have a program. Let us note here that in an obvious way, Turing machines can also be based on programs. This point of view is emphasized by Schönhage, cf. [Sch80] and [SGV94].

type (with obvious time measure) is real time equivalent to certain successor RAM types with uniform time measure.[3]

In many works on the complexity of computational problems arising in algorithmic number theory, cryptography and related areas, it seems to be assumed that the underlying model is on the one hand bit-oriented and on the other hand, storage access is more or less immediate. These requirements are met by successor RAMs with logarithmic time measure. The very limited instruction set of these models does however often not make it possible to obtain the claimed running times in a straightforward way. For example, very often the algorithms and their analyzes require the presence of commands for addition and subtraction of registers.

This situation motivates the search for a general result which transforms a result for any kind of random access machine model with "reasonable" time and space measures to a result for successor RAMs.

In this work, we give a rigorous definition of a RAM type with a very general instruction set (whose machines we call *RAM with extended instruction set*). With an adequate (and intuitive) time measure, we show that machines of this type can be simulated "quasi-optimally" (optimally up to "logarithmic factors") by successor RAMs with logarithmic cost measure. Concerning space complexity, the result is "quasi-optimal" too and in fact relates a particularly strong space measure on successor RAMs with a weak space measure for the general RAM type. The result on time complexity shows in particular that RAMs with additional commands for addition and subtraction and / or for concatination and shifts can be simulated "quasi-optimally" by successor RAMs with respect to the logarithmic cost function.

The simulation is straightforward, and in fact, it essentially already appeared in the literature before; cf. the proof of [WW86, Theorem 19.28]. However, an extensive search in the literature did not reveal a result as the one given in this work even for the simulation of RAMs with commands for addition and subtraction by successor RAMs. It is exactly the lack of a suitable reference which motivated the author to write this work.

Acknowledgment

I thank Pierrick Gaudry for discussions on computational models.

[3]The use of the notion "real time equivalent" in [Sch80] is different for its use at other places in the literature, e.g., in [vEB92] and [WW86]. In the spirit of [vEB92], one might say that these models *simulate each other in linear time*. With the definitions of [WW86], the *time measures of the models are linearly related.*

2 Basic definitions and observations

We assume that the reader is familiar with RAM models, at least on an intuitive basis. Briefly, a *successor RAM* type is a RAM type with only one arithmetic instruction: the computation of the successor.[4]

In [Sch80] two such types are defined, called RAM0 and RAM1. Let us recall the particular definitions in [Sch80] on a conceptual level and compare them to other definitions of RAM types in literature.

Let us define the set of *natural numbers* $\mathbb{N}$ as the set $\{0, 1, 2, \ldots\}$.

All machines defined in [Sch80] operate on the alphabet $\{0, 1\}$ for input and output. Each machine has an input and output tape, which are read respectively write only. Furthermore, they have a program based on some instruction set. The instruction sets contain instructions based on the codes **input**, **output**, **goto**, **halt**. The input instruction reads a bit from the input tape and – according to the bit – jumps to one of two labels. The output instruction prints a bit.

As usual for random access machines, a machine of type RAM1 has registers, an accumulator, instructions to load a fixed natural number and to load and store data directly and indirectly, and an instruction for comparison. All registers and the accumulator can store arbitrary natural numbers (or bit strings). The registers are indexed by natural numbers, and the accumulator is by definition not a register. Different from other random access machine types, the RAM1 type only has one "arithmetic" (or operational) instruction: the computation of the successor. The type RAM0 is similar to the type RAM1. The essential differences are that the type RAM0 does not have an instruction for indirect addressing but it has an additional address register instead.

Usually, random access machine types defined in the literature have additional "arithmetic instructions". The most cited type in the literature seems to be the one by Aho, Hopcroft, and Ullman ([AHU74]). This type has instructions for addition, subtraction, multiplication and division. Other types have instructions for bitwise OR, AND and XOR, and still others have an instruction for concatenation; cf. [WW86, Section 1.2].

Let a RAM0 or RAM1 Π be given, and let x be an input to Π. Then there are essentially two different definitions of running time of Π for x: the *uniform* and the *logarithmic* running time. With the uniform running time, each instruction executed is given the time 1. In order to define the logarithmic running time, we first define the *size* of a natural number n as 0 if $n = 0$ and $\lfloor \log_2(n) \rfloor + 1$ otherwise.[5] Now for the logarithmic running time,

[4] In [WW86] a machine type called successor RAM is defined which has instructions for computation of the successor and the predecessor. We do not follow this definition.

[5] This definition of the size of a number follows the definitions in [WW86]. In [vEB92]

each instruction not involving registers or the accumulator is given the time 1, and the instructions involving registers or the accumulator are given as time $1+$ the sum of sizes of the numbers in the accumulator or the registers in question involved (the accumulator for comparison and computation of the successor, the accumulator and one register for direct access and the accumulator and two registers for indirect access).

We define the *uniform* or *logarithmic time measure* for Π as the function on the natural numbers assigning to each natural number x the corresponding running time of Π upon input of x.

On the other hand, when we speak of the state of the machine *at a particular time*, we refer to the state after a particular number of operations has been executed, that is, after a particular uniform time has passed.

From a bit-oriented point of view, the logarithmic time measure is clearly the more adequate one. After all it really measures the bits involved in the execution of a particular instruction. Because of this, in the following we base our results for time complexity on this measure. We therefore call the logarithmic measure also the *time complexity* and denote it by T.

We have already mentioned that it is shown in [Sch80] that the two successor RAM types RAM0 and RAM1 are real time equivalent with respect to the uniform time measure. In fact, the simulation in [Sch80] reveals that they are also real time equivalent with the logarithmic time measure, thus it does not matter which type we choose. Let us – somewhat arbitrarily – define a *successor RAM* as a machine of type RAM1.

There are various measures of space complexity for RAMs defined in the literature. In this work, for successor RAMs we use three space measures S_1, S_2, S_3 which are again functions on the inputs and are defined as follows:

Let us fix a successor RAM Π and some input x to it. Let $R_{i,t}$ be the content of register R_i at time t. Let $u_t(i) := \mathrm{sgn}(\mathrm{size}(R_{i,t}))$, that is, $u_t(i)$ indicates if register i is used at time t. Further, let

$$\begin{aligned} b &:= \sup\{i \in \mathbb{N} \mid \text{register } R_i \text{ is used during the computation}\} \\ &= \sup\{i \in \mathbb{N} \mid \exists t \in \mathbb{N} \ : u_t(i) = 1\} \ . \end{aligned}$$

the size of 0 is by definition 1. The *logarithmic function* in [AHU74] is the same as the size in [vEB92]; see also Section 4.

We now define:

$$S_1(x) := \sup_{t\in\mathbb{N}} \sum_{i=0}^{\infty} \big(\mathrm{size}(R_{i,t}) + \mathrm{size}(i)\cdot u_t(i)\big)$$
$$S_2(x) := \sum_{i=0}^{\infty} \sup_{t\in\mathbb{N}} \big(\mathrm{size}(R_{i,t}) + \mathrm{size}(i)\cdot u_t(i)\big)$$
$$S_3(x) := \sum_{i=0}^{b} \sup_{t\in\mathbb{N}} \big(\mathrm{size}(R_{i,t}) + \mathrm{size}(i)\big)$$

Clearly,

$$S_1 \le S_2 \le S_3 \ .$$

Measure S_2 seems to be the most accepted measure in the literature; cf. [vEB92], [WW86]. Measures as S_1 and S_2 but without the term for the size of the register number are also often used in the literature. For example, in [AHU74] the corresponding variant of measure S_2 is used. From a bit-oriented point of view, measure S_2 is however more natural.

In contrast to the definitions in [WW86], the space measures are also defined for inputs for which the machine does not terminate. For inputs for which the machine terminates, the measure is always finite, for inputs for which the machine does not terminate it might be finite or infinite.

Let – as define above – T be the time complexity of Π. Then

$$S_3 \in O(T) \ .$$

Indeed, let us fix some input x upon which Π terminates. Wlog. we can assume that numbers > 0 are only loaded directly and indirectly (no *fixed* number $n > 0$ is loaded). Then the successors of $0, \ldots, b-1$ have to be computed. The logarithmic running time for this is $\sum_{i=0}^{b-1}(1 + \mathrm{size}(i)) \ge \sum_{i=0}^{b} \mathrm{size}(i)$. Furthermore, if $R_{i,t}$ is $\ne 0$, then at some time $s < t$, $R_{i,t}$ has to be stored in register R_i, and the logarithmic time needed for this is at least $\mathrm{size}(R_{i,t})$.

3 Discussion and further definitions

We now strive for a general result which transfers propositions on a type of random access machines with a very broad arithmetic instruction set to propositions for successor RAMs. The instruction set should contain all instructions of the successor RAM type and additional instructions which we call *higher arithmetic instructions*. These higher arithmetic instructions define partial functions from $\mathbb{N}^n$ to $\mathbb{N}$ for some n, as for example do the

usual addition and subtraction instructions.[6]

In order that one can obtain a transfer result as desired, clearly, the partial functions defined by the higher arithmetic instructions must be computable. A subtle question is then what time and space requirements one should charge for the execution of an instruction at a particular time.

We now describe the machines and the time and space measures we consider in detail.

First, we generalize the definition of successor RAM (i.e. RAM1) in the following way: We do not anymore have just one input tape but several input tapes. Correspondingly, the input instructions now take the following form:

$$\textbf{input}\ m, \lambda_0, \lambda_1$$

Here m is a natural number $\leq$ the number of input tapes, and as before λ_0, λ_1 are labels. The operation given by this instruction is as follows: One symbol is read from tape m and then according to the symbol being 0 or 1, the program is continued at label λ_0 or λ_1.

We call the resulting machine type *multi-inputtape successor RAM* type (*mi-successor RAM* type for short).

We now define a type of computational machines which we call *RAM with extended instruction set* as well as time and space measures on them.

The set of instructions of the new type has two parts. The first part consists of the instructions of the successor RAM model; we call these instructions *basic instructions*. The second part is given as follows: For each successor RAM P, we introduce an instruction c_P. We call these instructions *higher arithmetic instructions*. The *arithmetic instruction* are then the instruction for computation of the successor and the higher arithmetic instructions.

The syntactic requirements for a (program of a) RAM with extended instruction set are as for successor RAMs.[7]

Let now a (program for a) RAM with extended instruction set Π be given. Then the operation of Π is as follows: The basic instructions operate as usual. The operation of c_P for a successor RAM P is as follows: This instruction causes P to be executed in the following way. If P has n input tapes, P takes as input the content of registers $1, \ldots, n$ of Π. The output tape of P is the accumulator of Π. If P terminates, Π continues with the next instruction, as usual. If P does not terminate, Π does not terminate either.

[6]The instructions one usually considers in RAM models define functions, not only partial functions.

[7]One can (formally) define RAMs and programs of RAMs in such a way that a RAM and the corresponding program are (by definition) identical.

We define three time measures for such a machine Π.

- *simple uniform time* simply counts the number of instructions of Π.
- *extended uniform time* is defined as follows: The time for each basic instruction is 1, and the time for some instruction c_P is the uniform time needed for the execution of P with the inputs currently present in the respective registers of Π.
- *extended logarithmic time* is defined in exactly the same manner based on logarithmic time: The time for each basic instruction is measured in logarithmic time, and the time for c_P is the logarithmic time needed for the execution of P with the inputs currently present in the respective registers of Π.

It is extended logarithmic time which captures best the intuitive idea of a bit-oriented measure for this machine type, and therefore, similarly to above, we call this measure *time complexity* and denote it by T.

Let still some RAM machine with extended instruction set Π above be given, and let x be an input for Π. Let $i = 1, 2, 3$. The ith *basic space measure* of Π applied to x is defined as $S_i(x)$ above applied to Π; let us denote this measure by SB_i.

We define the 1st *space measure of the execution of some arithmetic instruction P at a particular (simple uniform) time* of Π as the first space measure applied to P and the corresponding input (present in the corresponding registers of Π). Now $S_1(x)$ is the supremum of $SB_1(x)$ and the first space measure applied to the executions of the arithmetic instructions.

The definition of the measures S_2 and S_3 is a bit more complicated: Let $i = 2, 3$. Let P be a successor RAM such that c_P occurs in (the program of) Π (it might occur several times). Then we define $S_{i,P}(x)$ as $S_i(x)$ above but with respect to all states of P for all executions of P during the execution of Π. Let $c_{P_1}, \ldots, c_{P_k}$ with distinct machines $P_1, \ldots, P_k$ be all instructions occurring in the (program of) Π. Then we define $S_i(x) := SB_i(x) + \sum_{j=1}^{k} S_{i,P_j}(x)$.

Again we have

$$S_1 \leq S_2 \leq S_3 \ ,$$

and it is not difficult to see that

$$S_2 \in O(T) \ .$$

However, there are machines for which it does not hold that $S_3 \in O(T)$. In fact, S_3 can be exponentially large with respect to T. For example, there exists a successor RAM E which computes 2^n in a time of $O(n)$. Now

using the instruction c_E, one immediately obtains a RAM with extended instruction set which upon input of $n \in \mathbb{N}$ stores 1 in register 2^n and then terminates. For this machine we have $T \in O(n)$ and $S_3 \geq SB_3 \geq 2^n$.

4 The result

In order to formulate the main result, it is convenient to use the following function, called *logarithmic function* in [AHU74].

Definition For some $n \in \mathbb{N}$, we define $l(n) := 1$ if $n = 0$ and $l(n) := \lfloor \log_2(n) \rfloor + 1$ otherwise.

Theorem *Let some RAM with extended instruction set* Π *be given. Then there exists a successor RAM* Π' *such that the following holds:*

Π' *terminates if and only if* Π *terminates, and the output of* Π' *is equal to the output of* Π. *Furthermore:*

Let T *be the time complexity of* Π *and* T' *the time complexity of* Π', *and let* S_1 *be the* 1st *space measure for* Π *and* S_3' *the* 3rd *space measure for* Π'. *Then*

$$T' \in O(T \cdot l(S_1)) \subseteq O(T \cdot l(T))$$

and

$$S_3' \in O(S_1 \cdot l(S_1)) \ .$$

We give the *proof* in two parts: We first only show the result for the case that Π is a successor RAM, and then we address the simulation of arbitrary RAMs with extended instruction set. Note that the first result is non-trivial because of the bound on the third space measure of Π'. The simulation for the first result contains the essential idea for the general result as well.

The result for successor RAMs

The simulation Let a successor RAM Π be given. We now describe the machine Π' used for the simulation.

A key idea for the simulation is to simulate the registers and the accumulator of Π in the following way: There are cells for data, and they always only contain 0 or 1. As a very naive approach to this idea, one might try to store the register cells in arrays. There are however some problems with this approach: First, how does one cope with "overflow" of arrays and second, how does one use indirect addressing in an efficient way? One possibility for

the second problem would be to try to transfer such an array into one register. But note that we do note have an addition command at our disposal, so it is unclear how to implement this idea in a sufficiently efficient way.

Rather than storing the data of one register of Π in an array, we store it in a linked list: Each node of the list contains two entries which are each stored in one register of Π': The first entry is a data element (being 0 or 1), and the second entry is the address of the next node.

In the same way, we simulate the accumulator of Π, and furthermore, also in this way, we implement an address register used for indirect addressing.

We use a binary tree to guarantee fast access to the simulated registers. The tree is as follows: Each vertex of the tree has at most two children, and the edges to the children are labeled with 0 or 1.[8] Let us assume that at some time t, register R_x of Π contains data $d > 0$, and let $x_k \cdots x_0$ be the binary expansion of x and $d_\ell \cdots d_0$ be the binary expansion of d. Then at the corresponding time of the simulation, there is a path from the root of the tree following the labels $x_0, \ldots, x_k$. The end of the path is the beginning of the linked list, and the data cells of the list contain $d_0, \ldots, d_\ell$. If on the other hand $d = 0$, there is no such path. (There might be a partial path in the tree but not a full path.) It is this tree structure which allows for efficient manipulation of data of Π.

During the operation, new vertices are inserted into the tree if some register is used which previously contains 0, and vertices are deleted if a register is set back to 0.

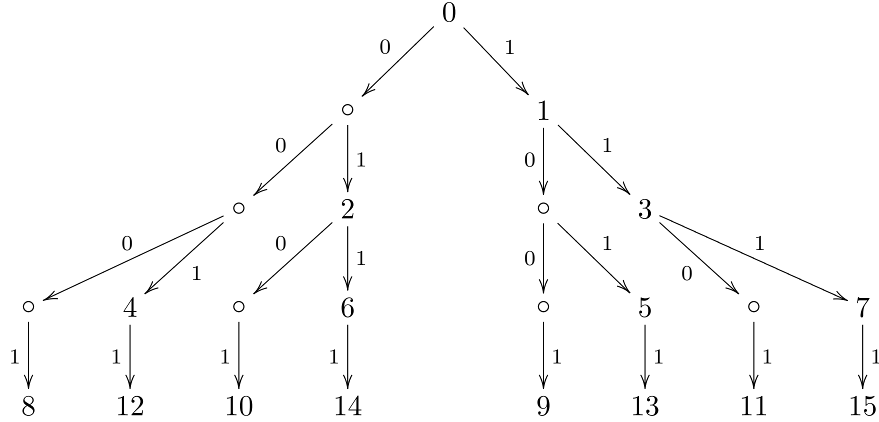
The structure just described is stored in the registers with even addresses, and one vertex occupies two consecutive even registers.

In order to access the storage efficiently, we use a stack and a counter. These are stored in the registers with odd addresses. Addresses of (tuples of) registers of Π' which were used for the tree or the data cells and are deleted are put onto the stack for reuse. (The stack is stored as an array, and each address occupies one register – as usual.) The counter stores the largest address used for the tree and the lists. If the stack is empty, the counter is incremented, and its value is used as an address.

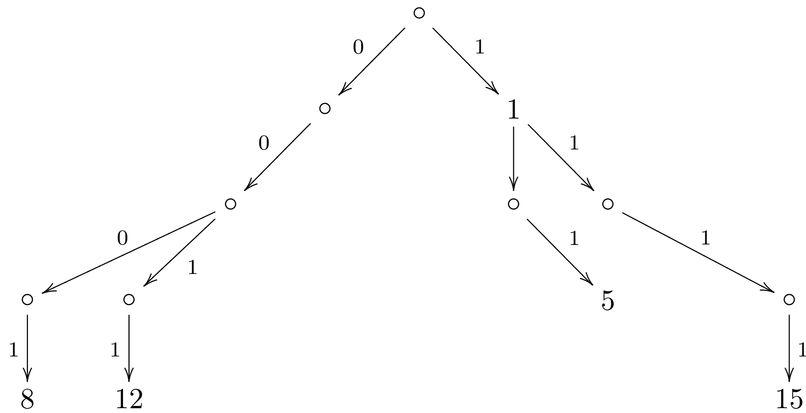
Illustration If for example all registers from 0 to 15 of Π are occupied, the tree looks like this. Here the edges with the numbers are the beginnings of the lists for the contents of the corresponding registers of Π. (The numbers

[8] We only use this labeling for the present informal description of the simulation. Only the children but not the labels are stored.

are not stored but only printed here for orientation.)



If only registers 8, 12, 5 and 15 are occupied, the tree looks as follows:



Again the numbers indicate that at these edges linked lists start.

If now, for example register, 15 is cleared, the corresponding edge as well as the two edges above are deleted, and the corresponding addresses of Π' are put on the stack for reuse.

The time and space requirements We now outline the results on time and space requirements obtained via the simulation.

The number of registers of Π' used for the simulation is in $O(S_1)$. Because of the use of the stack for storage management, the supremum of addresses used is in $O(S_1)$ as well. It follows that at during the whole computation the supremum of numbers stored in the registers is in $O(S_1)$.

Therefore $S_3' \in O(S_1 \cdot l(S_1))$.

Now, in order to load the content of the simulation of register R_i of Π at (simple uniform) time t into the simulation of the accumulator or the

simulated address registers we have to go along $O(l(R_{i,t})+l(i))$ nodes of the search tree and the linked list for the register.

Again, the supremum of numbers stored in a register of Π' used for the tree or the linked lists is in $O(S_1)$. This implies that the logarithmic time for such an operation is in $O((l(R_{i,t})+l(i))\cdot l(S_1))$.

Analogous considerations apply to the computation of the successor and comparison.

All in all, the time complexity of Π' is in $O(T\cdot l(S_1))$.

As already remarked, $S_1\in O(T)$ and thus $T'\in O(T\cdot l(S_1))\subseteq O(T\cdot l(T))$.

The general case

The outline for the general case is in fact nearly as the one for the restricted case. We however also have to simulate the arithmetic instructions c_P, and for this we use the simulation for successor RAMs just outlined.

We simulate the storage of Π' exactly as just described, but we only use registers R_i with odd i. The registers R_i with even i are then used to simulate the higher arithmetic instructions. If c_P is such a higher arithmetic instruction, then we simulate it by a successor RAM P' as described in the first part of the proof.

We now outline the results on time and space complexity for the general case. In fact, with minor modifications, the analysis in the special case still applies.

The number of registers of Π' used for the simulation of the registers, the accumulator, the stack, the counter and the address register of Π is now in $O(SB_1)$, and the supremum of addresses used for these structures is in $O(SB_1)$. Thus the supremum of numbers stored in any register of Π' used for these structures is in $O(SB_1)$ as well.

By the previous result, there exists a constant $C_1>0$ such that the supremum of addresses of Π' and the supremum of numbers stored in the registers of Π' used for the simulation of the arithmetic instructions is $\leq C_1S_1$.

All in all, we obtain $S_3'\in O(S_1\cdot l(S_1))$.

The logarithmic time for the simulation of loading or storing in registers of Π' is in $O((l(R_{i,t})+l(i))\cdot l(SB_1))$, and again we have an analogous result for the computation of the successor and comparison.

Furthermore, there exists a constant $C_2>0$ such that the simulation of any higher arithmetic instruction C_P of Π' at any particular time of Π' can be performed in logarithmic time $\leq C_2$ times the logarithmic time of the execution of P' with the particular input.

All in all, the time complexity of Π' is in $O(T\cdot l(S_1))$.

As already remarked, $S_1 \in O(T)$. □

5 Some further remarks

We now make some further remarks related to the Theorem.

- As usual, one also can define *non-deterministic* RAMs with extended instruction set. There are in fact two approaches: First, one can still leave instruction set as above (in particular, for each *deterministic* successor RAM P, we have a command c_P) but allow non-determinism in the same way as one usually does for RAM models. And second, one can in fact also extend the instruction set, also allowing instructions corresponding to non-deterministic successor RAMs. In any case, the Theorem in an obvious way also leads to results on the simulation of non-deterministic RAMs.

- As a variant of this, one can consider *randomized* RAMs. Here the same comments as above apply. In particular, we can use the Theorem to transfer propositions on the running times of Monte Carlo or Las Vegas algorithms. Propositions concerning Las Vegas algorithms are often formulated via expected running times in the following sense: For each input x the time complexity $T(x)$ is now a random variable, and one considers the function assigning to each input x the expected value of $T(x)$. Propositions on expected running times defined like this can then also easily be transferred. The same applies to propositions on space complexity with respect to the various measures.

- A usual RAM type is as the successor RAM types but with two arithmetic commands: addition and subtraction. (In [vEB92] and [WW86] this is called the *standard RAM*). Now, there exist mi-successor RAMs A and S which can perform addition and subtraction in linear time and with constant storage. Thus every such RAM immediately gives a RAM with extended instruction set with linearly related time complexity (with the logarithmic resp. extended logarithmic measures) and linearly related space complexities (with respect to the various measures). We can then apply the Theorem to obtain a "quasi-optimal" simulation by successor RAMs.

- If one substitutes logarithmic by uniform time and extended logarithmic by extended uniform time, the simulation does not lead to a "quasi-optimal" result. Indeed, let Π and Π' be as in the simulation, and let T_u and T'_u be the extended uniform resp. uniform time measures. If now Π is a successor RAM, the supremum of addresses used

and the supremum of values in any register are $\leq T_u$. One then obtains $T' \in O(T_u \cdot l(T_u))$. If however Π is some RAM with extended instruction set, one only has that the supremum of addresses used and the supremum of values in any register of Π are $\leq 2^{T_u}$. One then merely obtains $T'_u \in O(T_u^2)$.

- Of course, with RAMs with extended instruction set and the simple uniform time measure, one can obtain nearly arbitrarily small running times. Two special cases are however worthwhile mentioning:

  Let Π is a "standard RAM" with simple uniform time measure T_s. Then with the simulation we obtain a successor RAM Π with simple uniform time measure T'_s and $T'_s \in O(T_s^2)$. The argument for this is exactly as the one for the previous item. This result is given in [WW86, Theorem 19.28].

  However, if one allows all four arithmetic instructions, one obtains a dramatically different model; see [BMS85], [Sch79] and [WW86, Theorem 20.12], [WW86, Theorem 20.35]: The set of languages which can be recognized in polynomially bounded time on a nondeterministic machine can then also be recognized in polynomially bounded time on a deterministic machine and is equal to the set of languages which can be recognized in polynomially bounded space on a Turing machine. From a complexity theoretic point of view, this model can be considered as a parallel model.

- One can "iterate" the definition of the machine type "RAM with extended instruction" set by defining a new type which has as arithmetic all instructions of the RAM with extended instruction set. By iterating this procedure, we obtain a sequence RAM types indexed by the natural numbers; let us call any machine of these types a *RAM with iteratively extended instruction set*. We can now also iterate the definition of the extended logarithmic time measure and the space measures, obtaining in this way measures for all these machines. Let now such a RAM Π be given. Then one can also apply the simulation iteratively. Finally, one obtains a successor RAM Π' which simulates Π such that the following holds: With the notations as in the Theorem and the usual $\tilde{O}$-notation to capture logarithmic factors, we have:

  $$T' \in T \cdot \mathit{Poly}(\log(S_1)) \subseteq \tilde{O}(T) \text{ and } S'_3 \in \tilde{O}(S_1)$$

- It would be very interesting to have a general "quasi-optimal" result on the simulation of random access machines in some model by Turing machines. However, no such result is known. The following statement

is however obvious: Let Π be a RAM with extended instruction set. Then there exists a Turing machine (with 1-dimensional tapes) simulating Π with a time complexity of $O(T \cdot S_1) \subseteq O(T^2)$.

6 Summary

We give a summary of the definitions and results of this work on an intuitive level.

The starting point of this work is the observation that often the analysis of algorithms is performed in an ad-hoc way without reference to a specific model of computation. Implicitly however, the algorithms are usually analyzed in some kind of random access machine (RAM) model with some kind of instruction set. This motivates the search for a general transfer result to a truly bit-oriented model of computation. Such a result is given in this work.

Briefly, the result can be stated as follows: If one defines the time and space requirements of the instructions of the model in a bit-oriented way, one can obtain a transfer which is "quasi-optimal", i.e. "optimal up to a logarithmic factor".

Generally speaking, the result shows that if one employs the usual $\tilde{O}$ or O^* notation, it really is justified to take an intuitive and not too formal approach to complexity of algorithms.

Two aspects should however be added to caution the reader:

First, if one uses the O-notation and gives explicit "logarithmic terms", it really is necessary to first state the corresponding computational model. (At least as long as no stronger simulation result is known.)

Second, one might argue that a more adequate model of computation for algorithms with large space requirements is the multitape Turing model. There is however no general "quasi-optimal" transfer result from RAM models to the Turing model known.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[BMS85] A. Berton, G Mauri, and N. Sabadini. Simulations among classes of random access machines and equivalence among numbers succinctly represented. *Ann. Discrete Math.*, 25:65–90, 1985.

[Sch79] A. Schönhage. On the power of random access machines. In H. Maurer, editor, *Proc. 6th Internat. Coll. on Automata, Languages and Programming*, volume 71 of *LNCS*. Springer, 1979.

[Sch80] A. Schönhage. Storage Modification Machines. *SIAM J. Computing*, 9:490–508, 1980.

[SGV94] A. Schönhage, A. Grotefeld, and E. Vetter. *Fast algorithms – a multitape Turing machine implementation.* BI Wissenschaftsverlag, Mannheim, 1994.

[vEB92] P. van Emide Boas. Machine Models and Simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* Elsevier, 1992.

[WW86] K. Wagner and G. Wechsung. *Computational Complexity.* VEB Verlag der Wissenschaften, Berlin, 1986.

Claus Diem
Universität Leipzig
Mathematisches Institut
Johannisgasse 26
04103 Leipzig
Deutschland
diem@math.uni-leipzig.de