From RAM to SAT

NEU*

October 7, 2012

Common presentations of the NP-completeness of SAT suffer from two drawbacks which hinder the scope of this flagship result. First, they do not apply to machines equipped with random-access memory, also known as direct-access memory, even though this feature is critical in basic algorithms. Second, they incur a quadratic blow-up in parameters, even though the distinction between, say, linear and quadratic time is often as critical as the one between polynomial and exponential.

But the landmark result of a sequence of works overcomes both these drawbacks simultaneously! [HS66, Sch78, PF79, Coo88, GS89, Rob91]

The proof of this result is simplified by Van Melkebeek in [vM06, §2.3.1]. Compared to previous proofs, this proof more directly reduces random-access machines to SAT, bypassing sequential Turing machines, and using a simple, well-known sorting algorithm: Odd-Even Merge sort [Bat68].

In this work we give a self-contained rendering of this simpler proof.

For context, we note that the impressive works [BSCGT12b, BSCGT12a] give the stronger type of reduction where a candidate satisfying assignment to the SAT instance can be verified probabilistically in polylogarithmic time.

We work with the random-access Turing-machine model. Most other machines, including RAM, may be simulated in this model with only a polylogarithmic slow-down in time. On the other hand Turing machines are easier to describe and arguably cleaner.

Definition 1. A *random-access Turing machine* (RTM) M is a Turing machine with a constant number of tapes. These tapes are organized into k pairs. The ith pair, for $i \in \{1, \ldots, k\}$, consists of one RAM tape Ram_i and one register tape Reg_i. Each tape has its own head. M's transition function δ inputs the $2k$ symbols scanned and the state, and it outputs a new state, $2k$ new characters to be written on the tapes, and k head movements to adjacent cells for the register tapes only. M has k special, disjoint sets of *jump states*, denoted $J_1, \ldots, J_k$. When the machine leaves a state in J_i the following happens at once:

– (1) The head of Ram_i is moved to the cell whose address is on Reg_i,

*Work done as a group project for the Spring 2012 Graduate Theory of Computation class at Northeastern University. The authors are Zahra Jafargholi, Hamidreza Jahanjou, Eric Miles, Jaideep Ramachandran, and Emanuele Viola (instructor). We thank all of the students of the class for useful feedback during the preliminary stages of this project. Supported by NSF grant CCF-0845003.

– (2) the contents of Reg_i are erased, and
– (3) the head of Reg_i is moved to the beginning of the tape.
The machine starts with the input on the tape Ram_1.

We choose the terminology "register tape" because these tapes indeed hold registers when simulating RAMs.

Theorem 2 (From RAM to SAT)**.** *Let M be an RTM, and let $T = T(n) \geq n$ be a function computable in time $T^{O(1)}$ given any string of length n.*

Given an input x of length n one can construct a 3SAT instance ϕ of size $O(T \log^5 T)$ in time $|\phi| \cdot T^{O(1)}$ that is satisfiable if and only if there exists $y \in \{0,1\}^T$ such that M accepts (x, y) in $\leq T$ steps.

We note that the exponent of $\log T$ in the size of ϕ in Theorem 2 could be reduced easily, for example by using better sorting circuits. We also mention that the model of a non-deterministic machine can be obtained by viewing y in Theorem 2 as the machine's guesses. Finally, we note that the head on the tape Ram_1 containing the input may only be moved via jumps. Thus to recover some textbook algorithms for sequential Turing machines one either needs to simulate sequential access by random, or else first copy the input on a register tape. Alternatively one could equip RAM tapes with sequential-access capabilities, too.

Organization. In §1 we prove Theorem 2 under the additional assumption that the machine uses addresses of bounded length. We note that the machines resulting from standard simulations of other computational models such as random-access computers satisfy this assumption, see [GS89]. This proof relies on the possibility of sorting by a circuit of size $n \log^{O(1)} n$, a standard fact which for completeness is recalled in §3. In §2 we dispense with the assumption that the machine uses bounded addresses. In §4 we put everything together, in fact proving a stronger version of Theorem 2 where the 3SAT instance is computed more explicitly.

1 From bounded-address RTMs to 3SAT

In this section we prove our main Theorem 2 under the assumption that the machine is $O(\log T)$-bounded-address, as defined next.

Definition 3. Let $B(n)$ be a function. An RTM is *$B(n)$-bounded-address* if for every x of length n, and for every y, the computation of M on input (x, y) satisfies the following for each $i \leq k$: either the length of Reg_i is always bounded by $B(n)$, or else the machine never enters a jump state in J_i.

Theorem 2, under the assumption that the machine is $O(\log T)$-bounded-address, follows from the next two theorems. The first transforms a bounded-address RTM into a circuit satisfiability instance, and the second transforms the latter into a 3SAT instance.

Theorem 4. *Let $T = T(n) \geq n$ be a function computable in time $T^{O(1)}$ given any string of length n. Let M be an $O(\log T)$-bounded-address RTM.*

Given an input x of length n one can construct a Boolean circuit C of size $O(T \log^3 T)$ in time $|C| \cdot T^{O(1)}$ that satisfies the following: there exists $y \in \{0,1\}^T$ such that M accepts (x, y) in $\leq T$ steps if and only if there exists $y' \in \{0,1\}^{O(T \log T)}$ such that $C(y') = 1$.

Proof. Recalling Definition 3, we refer to tapes Reg_i which have length always bounded by $O(\log T)$ as *bounded register tapes*. Note that M has two other types of tapes, RAM tapes and unbounded register tapes; we are going to treat the last two types in a similar fashion.

We define a *configuration* of M to contain the following items:

- M's current timestep ($O(\log T)$ bits),
- M's current state ($O(1)$ bits),
- the entire contents of each bounded register tape including head position ($O(\log T)$ bits, using the fact that M is bounded-address),
- the head position and the content of the indexed cell of RAM and unbounded register tapes ($O(\log T)$ bits, using the fact that M is bounded-address and runs in time T).

Summing the bounds above, we see that the size of a configuration is $O(\log T)$.

We now describe the circuit C, see Figure 1 for a schematic representation. The input $y' \in \{0,1\}^{O(T \log T)}$ to the circuit is parsed in the following way. The first T bits contain y (the second input for M), and the rest contains T configurations $c_1, \ldots, c_T$ for the computation of $M(x, y)$. We check that these configurations encode an accepting computation of M, in two phases. In the first phase (comprising points 1, 2, and 3 below), we check the validity of states and head positions for all tapes, and we verify the consistency of bounded register tapes, assuming what is read on RAM and unbounded register tapes is correct. This is a simple comparison of adjacent configurations. In the second phase (Point 4 below), we check the consistency of read/write operations on RAM and unbounded register tapes, one tape at a time. For each tape, we sort the configurations by the head position on that tape and within each head position by timestep. Then we can check the consistency of read/write operations again by a simple comparison of adjacent configurations.

To emulate the fact that M starts with the string (x, y) on Ram_1, we create a set of "dummy" configurations $c_{-L}, \ldots, c_{-1}$ for $L := |(x, y)| = O(n + T) = O(T)$. In c_{-i}, the head position of Ram_1 is on the ith cell which contains the ith bit of (x, y), the timestep is say -1 for identification purposes, and all other fields are set to 0. These configurations are added to $c_1, \ldots, c_T$. (These dummy configurations are only relevant to the second phase for Ram_1; they are numbered from $-L$ to -1 because conceptually they correspond to an initial walk through the input (x, y).)

More formally, C operates as follows:

1. Check that c_1 is the initial configuration of M.

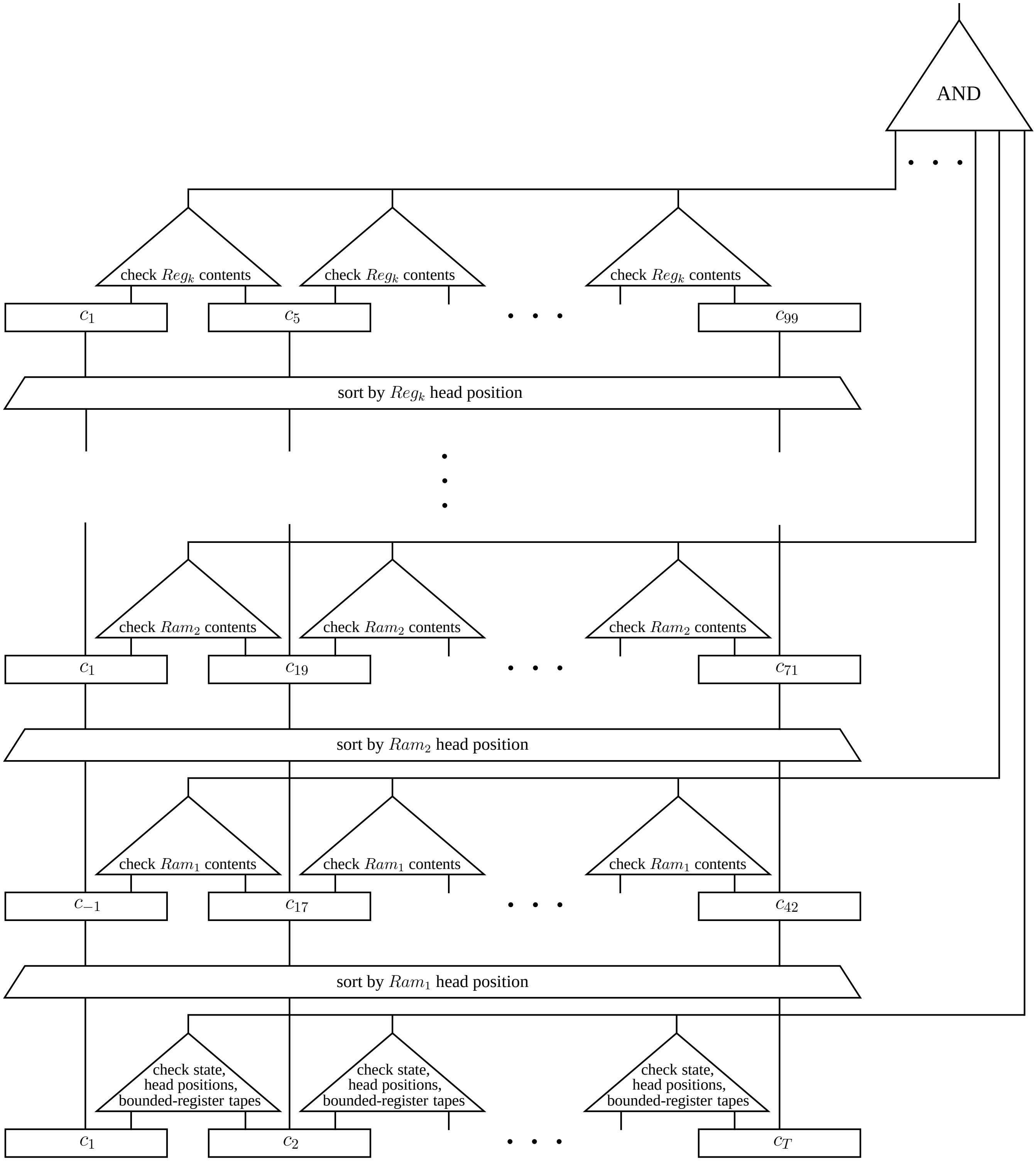

Figure 1: The circuit from Theorem 4

2. (States and heads) For each $t = 1, \ldots, T-1$, check that the state and $2k$ head positions in c_{t+1} are correct given c_t. Note that when c_t contains a state from a jump set J_i, this check verifies that Ram_i's head position in c_{t+1} matches the contents of Reg_i in c_t, and that the head on Reg_i is at the beginning.

3. (Contents of bounded-register tapes) For each $t = 1, \ldots, T-1$, check that the contents of bounded register tapes in c_{t+1} are correct given those in c_t. Note that when c_t contains a state from a jump set J_i, this check verifies that Reg_i's tape in c_{t+1} is all blank.

4. (Contents of RAM and unbounded register tapes) For each tape τ that is either a RAM tape or an unbounded register tape, sort the configurations (including the dummy configurations) increasingly by τ's head position and within each head position by timestep. Then for each adjacent pair of configurations $(c_t, c_{t'})$ where $c_{t'}$ is not a dummy configuration, check the following:

   – (a) If τ's head is at the same location on c_t and $c_{t'}$, check that the content of this indexed cell in $c_{t'}$ is correct given c_t (this also verifies the presence of the input on Ram_1, thanks to the dummy configurations),

   – (b) otherwise, check that the content of τ's cell indexed in $c_{t'}$ is blank.

5. If all checks pass and c_T contains M's accept state, output 1; else, output 0.

It can be shown that for every x there exist configurations that pass the above checks if and only if there is $y \in \{0,1\}^T$ such that M accepts (x, y).

We now complete the proof by observing that each step, and thus all of them, can be implemented by a circuit of size $O(T \log^3 T)$. Recall that the number of configurations, including dummy ones, is $O(T)$.

In Step 1, the initial configuration c_1 can be checked with size $O(\log T)$.

In Step 2, each check of two adjacent configurations is computable with size $O(\log T)$: the state check requires computing the $O(1)$-size transition function δ, and each head-position check requires computing δ and subtracting and testing equality of two $O(\log T)$-bit numbers. Thus the whole step is computable with size $O(T \log T)$.

In Step 3, for each bounded register tape Reg_i, each pair of adjacent configurations can be checked in size $O(\log T)$: it requires computing δ and testing equality of $O(\log T)$ cells each of size $O(1)$. (Here we may think of the head position as being encoded by marking the indexed tape cell.) Thus the whole step is computable with size $O(T \log T)$.

In Step 4, sorting the configurations for a tape τ can be performed in size $O(T \log^3 T)$ using the circuit constructed in Theorem 7. Then, checking each pair of adjacent configurations for τ can be computed in size $O(\log T)$: it requires computing δ and testing equality of two $O(\log T)$-bit head positions and two cells. Thus the whole step is computable with size $O(T \log^3 T)$. □

Theorem 5. *Given a Boolean circuit C of size T one can construct in polynomial time a $3SAT$ instance ϕ with $O(T)$ variables and $O(T)$ clauses such that ϕ is satisfiable if and only if there exists a string y for which $C(y) = 1$.*

Proof. We construct a SAT formula whose variables are the input y of C, plus one new variable for each gate in C. We assume without loss of generality that C contains only fan-in-1 NOT gates and fan-in-2 AND gates (any circuit can be represented in this form with only an $O(1)$ multiplicative increase in the number of wires). We construct a set of clauses for each gate in the circuit as follows:

For a NOT gate represented by variable g with input represented by w, add

$$(g \vee w \vee w) \wedge (\overline{g} \vee \overline{w} \vee \overline{w});$$

in any satisfying assignment $g =$ NOT w.

For an AND gate represented by variable g with inputs represented by w_1 and w_2, add

$$(w_1 \vee w_2 \vee \overline{g}) \wedge (w_1 \vee \overline{w_2} \vee \overline{g}) \wedge (\overline{w_1} \vee w_2 \vee \overline{g}) \wedge (\overline{w_1} \vee \overline{w_2} \vee g);$$

in any satisfying assignment $g = w_1$ AND w_2.

Finally, let ϕ be the $3SAT$ formula obtained from the conjunction of all the formulas for all the gates in C. It follows that there exists y such that $C(y) = 1$ if and only if ϕ is satisfiable. □

2 From general to bounded-address RTMs

Theorem 6 (From general to bounded-address RTMs)**.** *For any function $T = T(n) \geq n$, given an RTM M one can construct in time $|M|^{O(1)}$ an $O(\log T)$-bounded-address RTM M' that satisfies the following:*

for every input x of length n, there exists $y \in \{0,1\}^T$ such that M accepts (x, y) in T steps if and only if there exist $y \in \{0,1\}^T, y' \in \{0,1\}^{bT \log T}$ such that M' accepts $(x,(y,y'))$ in $O(T \log^2 T)$ steps, where b is a universal constant.

Proof. M' operates by consolidating the memory locations used by M into a continuous block of length $\leq O(T \log T)$, thus allowing each location to be referenced with an address of $\leq O(\log T)$ bits. To do this we require a method for mapping long addresses to short addresses. This mapping is done separately for each RAM tape, and we now describe it for one such tape.

We define a one-to-one correspondence between the set of all addresses of length $\leq T$ and the nodes of a full depth-$(T+1)$ binary tree in the following natural way. For an address A with $|A| \leq T$, the corresponding node is selected by walking from the root, choosing at the jth step either the right or left child depending if $A_j = 0$ or 1.

Fix an input $z = (x, y)$, and let $\{A_1, \ldots, A_r\}$ be the set of $r \leq T$ addresses accessed on the RAM tape by M on input z. The above set of random-access addresses corresponds to a subtree L where the ith address A_i corresponds to a node at depth $|A_i| + 1$.

In order to analyze the size of our subtree L, we observe that since each register tape is erased after a random access, the sum of the lengths of the addresses used during M's computation is at most T:

$$\sum_{i=1}^{r} |A_i| \leq T. \tag{1}$$

Further, since the ith address A_i corresponds to a node at depth $|A_i| + 1$, we have

$$\#\text{nodes in } L \ \leq \sum_{i=1}^{r} (|A_i| + 1) \leq T + r \leq 2T. \tag{2}$$

We represent L in a specific, compact fashion as a sequence of nodes augmented with the following auxiliary information: a left-child pointer, a right-child pointer and a flag. The flag indicates whether or not the node corresponds to an address A_i accessed by M. If set (to 1), the node stores two more items: the address A_i itself and the storage space for one cell. The address A_i is used to verify the integrity of L (i.e. that L does not use a single node for multiple addresses). The storage space is updated throughout the simulation with the value that M would have at address A_i. As pointers to the children require $O(\log T)$ bits, the amount of memory needed to store L is

$$\leq \sum_{i=1}^{\#\text{nodes}} O(\log T) + \sum_{i=1}^{r} (|A_i| + O(1)) = O(T \cdot \log T). \tag{3}$$

Now we describe how M' simulates M. This simulation involves copying various information among tapes. To facilitate such tasks, we equip M' with an additional constant number of tapes. We also note that copying one bit from a RAM cell typically costs a number of steps which is logarithmic in the address of the cell, due to the need to write down this address which is then erased, cf. Definition 1. One use of the additional tapes is to backup addresses used in jumps, to keep track of the heads on the RAM tapes.

The additional input y' to M' encodes k trees, $y' = (L_1, L_2, \ldots, L_k)$, where k is the number of RAM tapes of M. In an initialization step, M' sweeps through the whole input $(x, (y, y'))$, computes $n = |x|$ and $T = |y|$, and additionally for each node of L_1 with a set flag it does the following. If the address field A is $\leq |(x, y)|$, M' verifies that the storage field contains the corresponding bit of (x, y), otherwise it verifies it contains blank. M' also verifies that the storage fields of nodes on the other trees $L_2, \ldots, L_k$ all contain blank, and notes the address of the root of each L_i.

This initialization phase takes $O(|L| \log |L|) = O(T \log^2 T)$ time.

After the initialization phase, M' begins the simulation of M. When M leaves a jump state in a set J_i, M' intercepts the computation and does the following.

1. M' jumps to the address of L_i's root.

2. It traverses L_i according to the bits of the address A written on Reg_i. (Namely for $j = 1, \ldots, |A|$, if the jth bit of A is 0 then jump to the left child and otherwise jump to the right child.) Let N be the node reached by this process.

3. It verifies that N's flag is set, and that the stored address $= A$. It moves the head of Ram_i to the cell which holds the storage for A.

4. Finally, it erases Reg_i.

Accessing an address A by this process takes time $|A| \cdot O(\log^2 T)$, because for each bit of A the child pointer of length $O(\log T)$ is copied from a RAM tape in $O(\log T)$ steps per bit.

Including the initialization phase and the $\leq T$ steps not involving jumps, in total the simulation takes time at most

$$O(T \log^2 T) + T + \sum_{i=1}^{r} |A_i| \cdot O(\log^2 T) \leq O(T \log^2 T).$$

Finally, it can be seen from the construction that, for every x, if there exists a y such that M accepts (x, y), then there exist y and trees $L_1, ..., L_k$ for which M' accepts $(x, (y, L_1, ..., L_k))$. Conversely, if M' accepts $(x, (y, y'))$ for some y, y' then, as a result of the integrity check performed in Step 3 of the simulation, for each address A accessed in a tree L_i in M', there corresponds a unique address on Ram_i in M. Consequently, M also accepts (x, y). □

3 Sorting

In this section for completeness we show how to sort using a quasi-linear size circuit.

Theorem 7. *Given n and r one can construct in time $(rn)^{O(1)}$ a circuit of size $O(r \cdot n \cdot \log^2 n)$ that sorts any sequence of n r-bit elements by keys of $\leq r$ bits.*

We use a comparison-based sorting algorithm. The only difficulty is that to convert the algorithm into a circuit that implements it, we need the sequence of comparisons to be independent of the input. This is not the case with the most common algorithms. For example, the comparisons in the merge subroutine of Mergesort depend on the input.

We will use a variant of Mergesort, namely Odd-Even Mergesort [Bat68]. This is just like Mergesort except that the problematic merge subroutine is replaced with a subroutine whose comparisons do not depend on the input. We now present the algorithm. In the rest of this section we abbreviate Odd-Even by OE.

Algorithm 1, OE-MERGE(A, n), merges the two already sorted halves of the sequence $A = [a_0, a_1, \ldots, a_{n-1}]$, resulting in a sorted output sequence. It uses an operation CMP-EX$(A,\ i,\ j)$ that compares values at indices $i < j$ of the sequence A and exchanges them if and only if $a_i > a_j$. (CMP-EX is short for Compare-Exchange.) Algorithm 2, OE-MERGESORT, uses OE-MERGE as a subroutine.

Lemma 8. *OE-MERGESORT correctly sorts any input sequence of n elements.*

Algorithm 1 OE-MERGE(A, n)

Require: Sequence $A = [a_0, \ldots, a_{(n-1)}]$ of length n, such that $[a_0, a_1, \ldots, a_{n/2-1}]$ and $[a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$ are sorted; $n \geq 2$; n is a power of 2.

Ensure: Sequence A is replaced with its sorted version.

```
if $n = 2$ then
  CMP-EX($A$, 0, 1);
else
  OE-MERGE($[a_0, a_2, \ldots, a_{(n-2)}]$, $n/2$); //the even subsequence
  OE-MERGE($[a_1, a_3, \ldots, a_{(n-1)}]$, $n/2$); //the odd subsequence
  for $i \in \{1, 3, 5, 7, \ldots, n-3\}$ do
    CMP-EX($A$, $i$, $i+1$);
  end for
end if
```

Algorithm 2 OE-MERGESORT(A, n)

Require: $n \geq 1$; n is a power of 2

Ensure: Sequence A is replaced with its sorted version.

```
if $n > 1$ then
  OE-MERGESORT($[a_0, a_1, \ldots, a_{n/2-1}]$, $n/2$); //first half
  OE-MERGESORT($[a_{n/2}, a_{n/2+1} \ldots, a_{n-1}]$, $n/2$); //second half
  OE-MERGE($A$, $n$);
end if
```

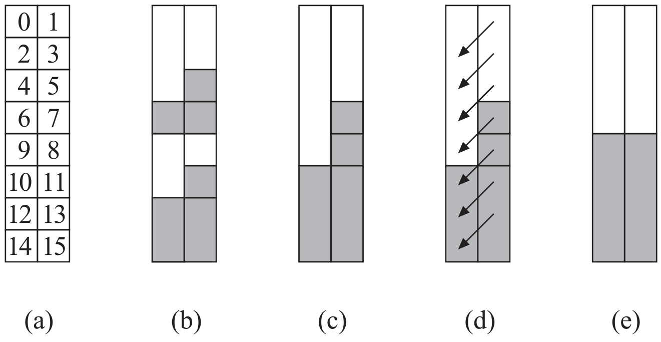

Figure 2: OE-MERGE illustration

Proof. Since OE-MERGESORT is a comparison-based sorting algorithm, by the so-called "0-1 Principle" it is sufficient to show that any sequence of 0s and 1s is sorted correctly. For completeness we sketch a proof of this principle in this paragraph. Let $A = [a_0, \ldots, a_{n-1}]$ be an input sequence of arbitrary numbers, and let $B = [b_0, \ldots, b_{n-1}]$ be the output sequence produced by the comparison-based sorting algorithm. If the algorithm fails to correctly sort A, then consider the smallest index k such that $b_k > b_{k+1}$. Define a function f such that $f(c) = 1$ if $c \geq b_k$ and $f(c) = 0$ otherwise, and let $f(B)$ be the sequence obtained by applying f pointwise to each element of B. Observe that $f(B)$ is not sorted. However it is easy to see that f commutes with any CMP-EX operation applied to any sequence X, i.e.,

$$f(\text{CMP-EX}(X, i, j)) = \text{CMP-EX}(f(X), i, j).$$

Hence

$$f(B) = f(\text{OE-MERGESORT}(A)) = \text{OE-MERGESORT}(f(A))$$

and so OE-MERGESORT fails to correctly sort the 0-1 sequence $f(A)$.

The correctness of OE-MERGESORT is immediate assuming the correctness of OE-MERGE. So we now argue the latter by induction on n, based on the recursive definition of OE-MERGE. Refer to Figure 2, reproduced here with permission from Lang's website [Lan01].

The base case $n = 2$ is clear. Assume that OE-MERGE correctly merges any two sorted 0-1 sequences of size $n/2$. We view an input sequence of n elements as an $n/2 \times 2$ matrix, with the left column corresponding to elements at the even-indexed positions $0, 2, \ldots, n-2$ and the right column corresponding to elements at the odd-indexed positions $1, 3, \ldots, n-1$ (Figure 2(a)).

Since the upper half of the matrix is sorted by assumption, the right column in the upper half has the same number or exactly one more 1 than the left column in the upper half. The same is true for the lower half (Figure 2(b)). Because each (length-($n/4$)) column in each half of the matrix is also individually sorted by assumption, the induction hypothesis guarantees that after the two calls to OE-MERGE both the left and right (length-($n/2$)) columns are sorted (Figure 2(c)).

At this point only one of 3 cases arises:

1) The odd and even subsequences have the same number of 1s.

2) The odd subsequence has a single 1 more than the even subsequence.

3) The odd subsequence has two 1s more than the even subsequence.

In the first two cases, the sequence is already sorted. In the third case, the CMP-EX operations (Figure 2(d)) yield a sorted sequence (Figure 2(e)). □

To conclude the proof of Theorem 7 we only need to argue efficiency. Let $S_M(n)$ denote the number of CMP-EX for OE-MERGE for an input sequence of length n. We have the recurrence $S_M(n) = 2 \cdot S_M(n/2) + (n/2 - 1)$, which yields $S_M(n) = O(n \cdot \log n)$. Let $S(n)$ denote the number of calls to CMP-EX for OE-MERGESORT with an input sequence of length n. Then we have the recurrence $S(n) = 2 \cdot S(n/2) + (n \cdot \log n)$, which yields $S(n) = O(n \cdot \log^2 n)$.

Finally, we observe that a comparator for two r-bit strings based on any key of length $\leq r$ can be implemented by a Boolean circuit of size $O(r)$, and thus the entire sorting network can be implemented by a Boolean circuit of size $O(r \cdot n \cdot \log^2 n)$, establishing Theorem 7.

4 Putting things together

In this section we state a more explicit version of our main Theorem 2. We then prove the original version as well as the more explicit one.

In the next theorem we make the additional restriction that the time bound $T(n)$ is computable in time $\log^{O(1)} T(n)$. This holds for most natural functions.

Theorem 9 (From RAM to SAT, more explicit)**.** *Let M be an RTM, and let $T = T(n) \geq n$ be a function computable in time $\log^{O(1)} T$. Given an input x of length n and an index $i \leq O(T \log^5 T)$ one can compute in time $\log^{O(1)} T$ the i-th clause of a 3SAT instance ϕ (of size $O(T \log^5 T)$) that is satisfiable if and only if there exists $y \in \{0,1\}^T$ such that M accepts (x, y) in $\leq T$ steps.*

Proof of Theorems 2 and 9. We first note the size of the 3SAT instance produced by combining the preceding theorems. By Theorem 6 we assume we have an equivalent RTM M' that is $O(\log T)$-bounded address and runs in time $T' := O(T \log^2 T)$. From M', Theorem 4 produces a circuit C of size $O(T' \log^3 T') = O(T \log^5 T)$. Finally from C, Theorem 5 produces the 3SAT formula ϕ which is also of size $O(T \log^5 T)$. This establishes Theorem 2.

We now observe that, due to the regularity of the circuit produced by Theorem 4, each clause of ϕ can be computed in time $\log^{O(1)} T$. Note that T' is computable in time $\log^{O(1)} T$ under the assumption that T is. Also, the time $\log^{O(1)} T$ subsumes that needed to write down the description of M' from M (Theorem 6). (Actually in our theorem statements the algorithm in the conclusion may depend on M, in which case we could just assume we have M' at our disposal. But more generally M can be given as input together with x and i.)

Let C be the circuit produced from M' by Theorem 4, and note that each clause of ϕ corresponds to exactly one gate in C. We view C as being composed of a series of subcircuits, where each subcircuit either checks the consistency of two adjacent $O(\log T')$-bit configurations, or sorts the set of configurations by the head position of one of the tapes and within head position by timestamp (refer to Figure 1).

Given an n-bit input x and an index i, we first determine which subcircuit the ith clause belongs to. If the subcircuit is a sorting circuit constructed via Theorem 7, then we compute the clause in time $\log^{O(1)} T$ exploiting the simple structure of this circuit. Otherwise, the subcircuit checks two adjacent configurations; such a circuit has size $O(\log T')$, and can also be constructed in this time. So in this case we may explicitly compute the subcircuit and output the corresponding clause. □

Acknowledgments. We thank Eli Ben-Sasson for a stimulating conversation on [BSCGT12b, BSCGT12a]. We are also grateful to Dieter van Melkebeek for pointing out his proof [vM06, §2.3.1] to us (alas, after this work was first posted).

References

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BSCGT12a] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. *IACR Cryptology ePrint Archive*, 2012:71, 2012.

[BSCGT12b] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:45, 2012.

[Coo88] Stephen A. Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26(5):269–270, 1988.

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[HS66] Fred Hennie and Richard Stearns. Two-tape simulation of multitape turing machines. *J. of the ACM*, 13:533–546, October 1966.

[Lan01] Hans Werner Lang. Odd-even mergesort, 2001.
`http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oem.htm`.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, 1979.

[Rob91] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *J. of the ACM*, 25(1):136–145, 1978.

[vM06] Dieter van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2(3):197–303, 2006.