198:538 Lecture Given on February 23rd, 1998

Lecturer: Professor Eric Allender
Scribe: Sunny Daniels

November 25, 1998

1 A Result

From theorem 5 of the lecture on 16th February 1998, together with the obvious result that DSPACE($t(n)$) $\subseteq$ NSPACE($t(n)$), we have:

Theorem 1 *For all running-time functions $t(n)$ with $t(n) \geq \log n$ for all $n \in \mathbb{N}$, we have:*

$$\begin{aligned} ATIME(t(n)) &\subseteq DSPACE(t(n)) \\ &\subseteq NSPACE(t(n)) \\ &\subseteq ATIME(t(n)^2) \end{aligned}$$

2 Computation in Sub-Linear Time

2.1 Problem with our Current Definition of Turing Machine

In this course, we will find it useful to be able to define Turing Machines whose running time is sub-linear in their input length. With our current definition of a Turing Machine, a Turing Machine that reads both of any two input squares x and y must step through all of the squares between x and y, one square at a time. This means that a sub-linear-time Turing Machine can only look at a substring of its input of length proportional to its running time. This means that many problems that can be solved in, for example, logarithmic time on real computers can not be solved in logarithmic time on a Turing Machine, according to our current definition of a Turing Machine.

2.2 Modified definition of a Turing Machine

In order to obtain a more useful definition of sub-linear-time computation on a Turing Machine, we will slightly redefine our notion of a Turing Machine. Instead of having an input tape, our redefined Turing Machine will have a random-access input array, together with an address tape. The address tape is accessed sequentially, in the same manner as the work tapes, but is write-only. We assume that the tape alphabet used on the address tape contains the symbols “0” and “1”, which are used to form binary representations of non-negative integers. The Turing Machine will have, in addition to the usual operations of a Turing Machine, an “input read” operation. This operation will look at the address tape to determine the non-negative integer k whose binary representation is stored on the address tape. (If the address tape does not contain the binary representation of a non-negative integer when “input read” is called, we assume that the Turing Machine hangs, or performs some undefined action. It is the responsibility of the Turing Machine “programmer” to ensure that the address tape always contains the binary representation of a non-negative integer when the “input read” operation is called.) The new state of the finite state control unit will then be determined by both the current state of the control unit, and the symbol in the given input location. The overall layout of our “new” Turing machine is shown in figure 1.

2.3 Important Note

It is important to note that we assume that the “input read” operation does not destroy or modify the contents of the address tape. Therefore, the Turing Machine can perform a sequence of input reads without having to re-write the whole input address to the tape before each read. For example, the Turing machine can write “1100101” to the tape (and end up with its head scanning the initial “1”), and then perform an “input read” operation to read address 1100101. The machine can then, without moving its head, change the address to “0100101” and perform another “input read” operation to read address 100101. The machine can then move its head one square to the right, change the initial “1” to a “0”, perform another “input read” to read address 101, and so on.

3 Circuit Families

Definition 1 *A* circuit family *is a set:*

$$\mathcal{C} = \{C_n : n \in \mathbb{N}\}$$

where each C_n is a circuit with n inputs.

Here, a “circuit” means a combinatorial logic circuit, composed of logic gates, with exactly one output. We stipulate that:

1. The fan-in of the logic gates can be either bounded (by a constant) or unbounded, depending upon the application.
2. The fan-out of the logic gates can be either bounded (by a constant) or unbounded, depending upon the application.
3. Each logic gate is either an *and gate*, an *or gate*, or a *not gate*.

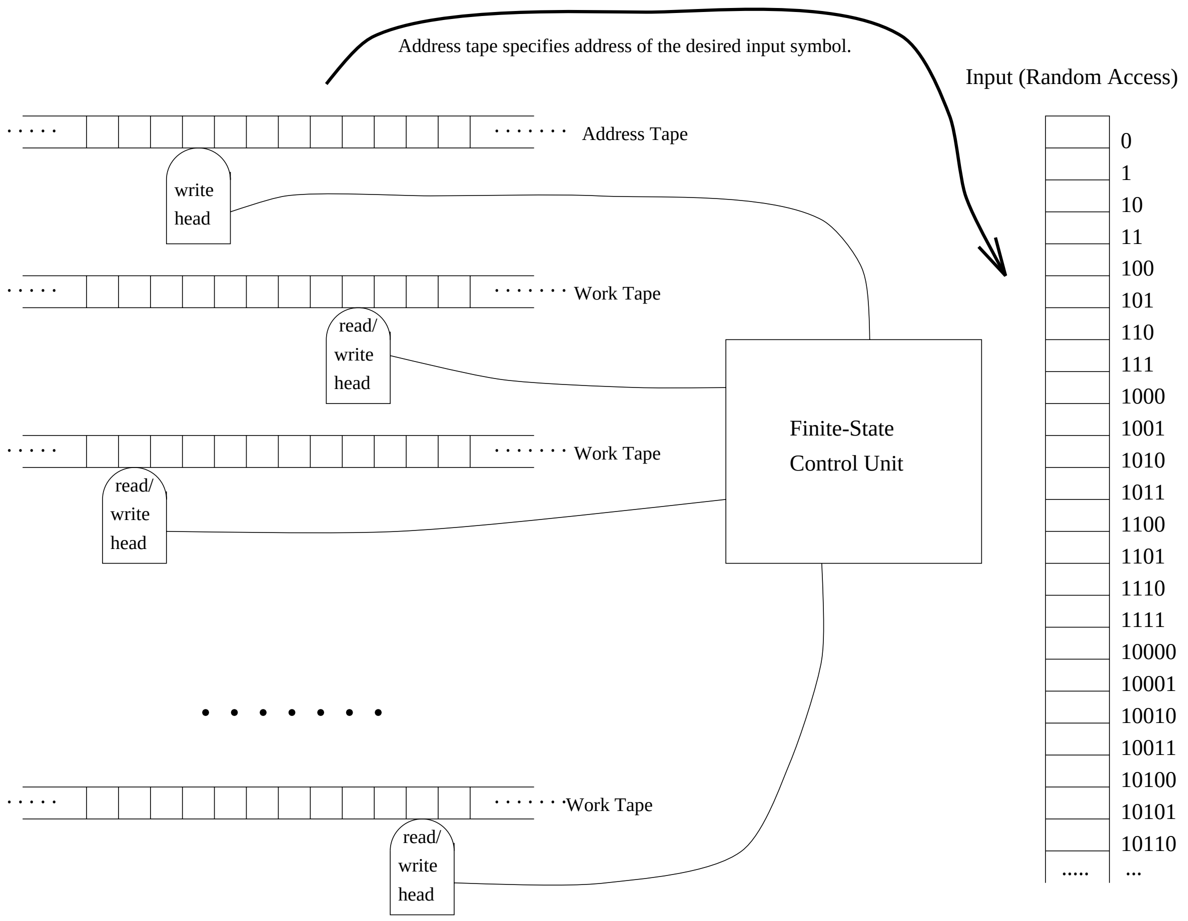

Figure 1: A Turing Machine with Random Access to its Input

4. Some ordering $i_1, i_2, \ldots, i_n$ is assigned to the inputs of each circuit C_n in $\mathcal{C}$.

5. The *size* of a circuit C_n is the number of gates in C_n, plus the number of inputs to C_n. We will sometimes call the inputs to a circuit C_n "input gates".

6. The *size* of a circuit family $\mathcal{C}$ is a function from $\mathbb{N}$ to $\mathbb{N}$ which maps each natural number n to the size of C_n.

3.1 Acceptance of a Language by a Circuit Family

The language $L(\mathcal{C})$ accepted by a circuit family

$$\mathcal{C} = \{C_n : n \in \mathbb{N}\}$$

is defined as follows:

1. For any bit string $x \in \{0,1\}^*$, we feed the bit in each bit position i of x (for every $i \in \{1, 2, \ldots, n\}$) into the ith input of $C_{|x|}$.

2. If the resulting output of $C_{|x|}$ is 1, then we define $x \in L(\mathcal{C}_{|x|})$.

3. If the resulting output of $C_{|x|}$ is 0, then we define $x \notin L(\mathcal{C}_{|x|})$.

4 Every Language has a Circuit Family of Size $2^{O(n)}$

One unusual property of circuit families, as a model of computation, is that *every language* (over the alphabet $\{0,1\}$) is accepted by some circuit family: even the undecidable languages!

Theorem 2 *For* every $L \subseteq \{0,1\}^*$, *there is a circuit family* $\mathcal{C} = \{C_n\}$ *accepting* L *s.t. the size of each circuit* C_n *is no greater than* 2^{2n}.

Proof: For each natural number n, we will define the circuit C_n as follows.

1. For each number $m \in \{1, 2, \ldots, n\}$, we construct an input i_m to C_n. We connect a *not* gate to each input i_m to produce the inverted input $\overline{i_m}$.

2. For each $x \in L$, we add an *and* gate $\wedge_x$ to C_n. The inputs of $\wedge_x$ are connected to the following places:

   (a) For each bit position m of x whose value is 0, we connect an input of $\wedge_x$ to $\overline{i_m}$.

   (b) For each bit position m of x whose value is 1, we connect an input of $\wedge_x$ to i_m.

It is obvious from the construction so far that the output of each such gate $\wedge_x$ is 1 when the input to the circuit C_n equals x, and 0 otherwise.

3. We then add a gate $\vee_{C_n}$ to the circuit C_n. The inputs to this gate are the outputs of the gates $\wedge_x$ for all of the strings $x \in L$.

4. It is now obvious that the output of the gate $\vee_{C_n}$ equals 1 when the input x to C_n is in L, and 0 otherwise. Hence, if we make the output of $\vee_{C_n}$ the output of the circuit C_n, then our circuit family will recognize the language L.

4.1 Proof of Upper Bound on Number of Gates

Now, any circuit C_n constructed in the above way will have n inputs, one *or* gate (namely $\vee_{C_n}$), and at most 2^n *and* gates (at most one for each of the 2^n strings in $\{0,1\}^n$). Hence, the size of C_n will, in general, be at most:

$$2^n + n + 1$$

It is easy to verify, by simple algebra, that $2^n + n + 1 \leq 2^{2n}$ for all positive integers n. Hence, each circuit C_n constructed in the above way will have at most 2^{2n} gates. This completes the proof of theorem 2.

4.2 Examples of This Construction

This construction is identical to the method, given in any textbook on elementary computer architecture, for constructing a combinatorial logic circuit to compute an arbitrary boolean expression. For examples of this construction, the reader can consult almost any textbook on elementary computer architecture, e.g. Tanenbaum [**?**].

5 Uniformity of Circuits

As remarked in the last section, even provenly undecidable languages can be decided by circuit families. Clearly, however, the circuit families for such provenly undecidable languages must be impossible to construct by any algorithmic means. Otherwise, we could construct an algorithm that constructs the appropriate-sized member of such a circuit family for its input length, and then simulates this circuit on the given input. This algorithm would hence decide a provenly undecidable language!

This point illustrates the need to consider, when reasoning about a circuit family, not only the *size* of the circuit family, but also the difficulty of constructing the member of the circuit family for any given input size.

5.1 Definition of Uniformity

Definition 2 *We say that a circuit family*

$$\mathcal{C} = \{C_n : n \in \mathbb{N}\}$$

is uniform *if there is a Turing machine that takes any $n \in \mathbb{N}$ as input and outputs some intuitively simple representation of C_n.*

As well as considering the *computability* of the circuit family, we will also consider the *complexity* of the problem of computing the circuit family:

Definition 3 *For any given complexity class $\mathcal{X}$, we say that a circuit family*

$$\mathcal{C} = \{C_n : n \in \mathbb{N}\}$$

is $\mathcal{X}$-uniform if there is a Turing machine whose language is of complexity $\mathcal{X}$ *that takes any $n \in \mathbb{N}$ as input and outputs some intuitively simple representation of C_n.*[1]

6 A Theorem about P-Uniformity

Theorem 3 *For any language L over the alphabet $\{0, 1\}$, the following two statements are equivalent:*

1. *L is in P.*
2. *L is recognized by some P-uniform circuit of polynomial size.*

We will prove the two halves of the equivalence separately:

6.1 Proof that 2 implies 1:

If L is recognized by a P-uniform circuit of polynomial size, then it is obviously trivial to build a polynomial-time Turing machine to recognize P. All this Turing machine has to do is:

1. Simulate the polynomial-time Turing machine that builds the circuit for L.
2. Simulate the circuit for L on the given input, to determine whether or not the given input is in L. Since the circuit is of polynomial size, this can be done in polynomial time.

Hence, assertion 2 implies assertion 1, as required.

[1]We will, however, use a different definition for the special case of DLOGTIME-uniformity. We will give this special definition in section 7 below.

6.2 Proof that 1 implies 2:

Take any language L that is in P. Then, by theorem 5 of the lecture given on 16th February 1998, L is in ASPACE($\log n$). Now, we will construct the circuit family:

$$\mathcal{C} = \{C_n : n \in \mathbb{N}\}$$

as follows. In the lecture given on Jan 26th, we showed that, if a deterministic Turing machine's running time is more than a particular value — this particular value is exponential in the machine's space bound — then the machine will never halt. It is fairly easy to apply the same reasoning to an alternating Turing Machine, and conclude that, if an alternating Turing machine's running time is more than this same value that is exponential in the machine's space bound, then the machine will never halt.

Hence, we can assume without loss of generality that the language L is accepted by a logarithmic-space Turing machine M_i whose running time is exponential in its space bound. Since M_i's running time is exponential in its space bound, which is in turn logarithmic in its input length, we know that M_i's running time is polynomial in its input length n. Let $p(n)$ be the polynomial function of n that bounds M_i's running time. Let $l(n)$ be the logarithmic function of n that bounds M_i's working-tape space.

Now, for each input length n, let the circuit C_n have exactly one "gadget"[2], whose name is the pair (x, m), for each sequence x of members of M_i's tape alphabet with $|x| \leq l(n)$ and number $m \in \{0, 1, 2, \ldots, p(n)\}$. We will fix some intuitively-simple means of representing the contents of M_i's working tapes by such a string x. (In the case of a multi-tape Turing machine, we may have to use additional symbols in our string x to mark the boundaries between the representations of the contents of different tapes.)

Let the output gadget of the circuit be labeled with the pair $(y, 0)$, where y is the string of tape-alphabet members representing the initial configuration of M_i's working tapes.

Now, we will choose the types of the gadgets, and connect the gadgets together, as follows:

1. For any pair (x, m), if the configuration represented by x does not correspond to an existential choice operation, universal choice operation or input-tape read operation, then the corresponding gadget is simply a wire to the gadget corresponding to $(y, m+1)$. Here, y is the (uniquely determined by x) configuration that M_i moves to from the configuration represented by x.

2. For any pair (x, m), if the configuration represented by x reads a bit of the input tape, then the gadget corresponding to (x, m) is defined as follows. Let y be the configuration that M_i moves to if the input bit equals 0. Let z be the configuration that M_i moves to if the input bit equals 1. Then,

[2]By "gadget", we mean a small finite combinatorial circuit made up of gates, whose size *does not* depend upon n.

the gadget corresponding to (x, m) will have three inputs, from the gadget corresponding to $(y, m + 1)$, the gadget corresponding to $(z, m + 1)$ and the input wire corresponding to the input bit being read in configuration x. Let's call the boolean values of the logic levels on these wires α, β and γ, respectively. Then the gadget corresponding to (x, m) is the gadget that computes the Boolean expression:

$$(\overline{\gamma} \wedge \alpha) \vee (\gamma \wedge \beta)$$

3. We assume that our definition of an alternating Turing Machine is such that no configuration both looks at the input tape and performs a universal or existential choice.

4. For any pair (x, m), if the configuration represented by x performs an existential choice, then the gadget corresponding to (x, m) is an *or* gate. The two inputs of this gate are connected to the outputs of the gadgets corresponding to $(y, m+1)$ and $(z, m+1)$, where y and z are the configurations corresponding to the two outcomes of the existential choice. (We assume that our definition of an alternating Turing Machine is such that all existential-choice steps have exactly two outcomes.)

5. For any pair (x, m), if the configuration represented by x performs a universal choice, then the gadget corresponding to (x, m) is an *and* gate. The two inputs to this gate are connected to the outputs of the gadgets corresponding to $(y, m+1)$ and $(z, m+1)$, where y and z are the configurations corresponding to the two outcomes of the universal choice. (We assume that our definition of an alternating Turing Machine is such that all universal-choice steps have exactly two outcomes.)

6. For any pair (x, m), if x corresponds to a halted configuration, then the gadget corresponding to (x, m) is defined as follows. If x corresponds to an accepting configuration, then the gadget corresponding to (x, m) is a logic input "hardwired" to the logic value 0 (i.e. false). On the other hand, if x corresponds to a rejecting configuration, then the gadget corresponding to (x, m) is a logic input "hardwired" to the logic value 1 (i.e. true).

It should be fairly clear from these definitions that the circuit "simulates" the machine M_i, in that the logic value coming out of the output gadget is 0 if M_i rejects the input on C_n's input wires, and 1 if M_i accepts the input on C_n's input wires.

It is now trivial to construct the machine M_j that builds the circuit C_n. All this machine has to do is to cycle through all possible pairs (x, m), where x is a string of length $l(n)$ and m is a natural number between 1 and $p(n)$; for each such pair, the machine applies the above rules to determine the type of the gadget and the source of its input wires, and outputs this information. Clearly, from the statements of these above rules, the computation for each such pair (x, m) can be performed in time polynomial in the input length n, and space logarithmic in the input length. Since the number of such pairs is polynomial in the input length, this means that the time required for the whole construction is only polynomial in n. This completes the proof of theorem 3.

It is also now obvious that the whole computation can be performed in space logarithmic in the input length. Hence, the circuit family $\mathcal{C}$ is also DLOGTIME-uniform.

7 Special Definition for case of DLOGTIME-uniformity

The definition of uniformity given above in section 5.1 does not apply to the special case of DLOGTIME-uniformity. We will only define DLOGTIME-uniformity for the case in which the fan-in of the gates is two. We define DLOGTIME-uniformity as follows:

Definition 4 *We say that a circuit family*

$$\mathcal{C} = \{C_n : n \in \mathbb{N}\}$$

is DLOGTIME-uniform if, for each of the following three languages, there is a deterministic k-tape Turing Machine that accepts the language in linear *time:*

1. *The set of all triples (n, g, t) for which g is a gate in C_n and g has type t ($t \in \{and, not, or, etc\ldots\}$).*
2. *The set of all triples (n, g, h) for which g and h are gates in C_n and $g \to h$.*
3. *The set of all quadruples (n, g, h, p) for which:*
   - *(a)* $p \in \{L, R\}^*$
   - *(b)* $|p| \leq \log_2 n$
   - *(c) Starting at h, and going back along path p, you reach g. Here "L" means "go from the current gate γ in the path to the gate whose output is connected to the left-hand input of γ". Similarly, "R" means "go from the current gate γ in the path to the gate whose output is connected to the right-hand input of γ".*