CSE 200 Lecture Notes
Turing machine vs. RAM machine vs. circuits

Chris Calabro

October 26, 2006

1 RAM model

There are many possible, roughly equivalent RAM models. Below we will define one in the fashion of a RISC computer and then show how to efficiently simulate common additional features. A *RAM machine* is a tuple $R = (k, \delta)$ where $k \in \mathbb{N}$ is the number of variable registers, and δ is a finite program. A *program* δ is a finite sequence of statements. A *statement* is one of

- $R_i \leftarrow M[R_j]$ (load)
- $M[R_i] \leftarrow R_j$ (store)
- $R_i \leftarrow R_i - R_j$ (subtract)
- $R_i \leftarrow R_i$ div 2 (shift right)
- $R_i \leftarrow 1$ (load constant 1)
- if $R_i \geq 0$, goto L (conditional jump)
- accept
- reject

where $i, j \in [k]$ and $L \in [|\delta|]$. (We could formalize a statement as a triple containing an opcode and the indexes of the operand registers or immediate constants, but little would be gained.) With these operations it is easy to see that we can simulate in $O(1)$ time each of the following: load the constant 0; addition; unconditional jump; jump if equal, unequal, less than, greater than, less than or equal; compute $R_i \mod 2$; shift left.

Since we are mostly concerned with the performance of a machine only up to a constant factor, we may assume that our machine has these capabilities when we want to stress the power of the machine, but may assume it only has the original instruction set when we want to stress the limitations of the machine, e.g. if we are trying to simulate an arbitrary RAM machine.

The total state of the machine at any given time is the values in the registers, the location of the program counter, and the values in the memory cells - formally a tuple $(R_1, \ldots, R_k, c, M)$ where each $R_i \in \mathbb{Z}$, $c \in [|\delta|]$, and $M : D \to \mathbb{Z}$ where $D \subseteq \mathbb{Z}$ is finite. The initial state is $(x, 0, \ldots, 0, 1, \emptyset)$ where $x \in \mathbb{N}$ is the input. If the state of the machine at time t is $(R_1, \ldots, R_k, c, M)$, then either the cth statement in δ is accept or reject, in which case the machine halts, or the state of the machine at time $t + 1$ is the result of executing the cth statement of δ, which at this point the reader should be able to formalize.

The *language* of R is the set of decodings of natural numbers that R accepts. A natural encoding scheme is to fix a (strictly) positive integer for each letter in the alphabet and represent a string by 'concatenating' together the encodings of each individual letter. Formally, fix some map $f : \Sigma \to \mathbb{Z}^{\geq 1}$ and extend f to a homomorphism $f^* : \Sigma^* \to \mathbb{Z}^{\geq 0}$ by

$$f^*(x_0 \cdots x_{n-1}) = \sum_{i=0}^{n-1} f(x_i) 2^{mi},$$

where $m = \lfloor \lg \max_{a \in \Sigma} f(a) \rfloor + 1$ is the length of the binary encoding of a single letter. An alternative is to assume $\Sigma = \{0, 1\}$ and just tack a leading 1 onto the string to encode it. Choose your favorite scheme.

Notice that multiplication is not in our instruction set. Nor is writing a register in a bit whose index is given by another register. In general, these operations cannot be simulated in a number of steps polynomial in the bit size of the input.

1.1 Simulation on a 1-tape TM

We claim that we can simulate t steps of $R(x)$ on a 1-tape TM T in time $O(t^3(n + t)^2)$. To initialize T, we decode x (by padding the left with a 1 bit, or whatever scheme we chose above) followed by k # symbols. These will delimit our representations of the k registers and the memory cells. This takes $O(n)$ time. The finite program of R can obviously be encoded in the transition function of T, the details of which would be extremely tedious but not very enlightening. The (finitely many) memory cells of R that contain nonzero values will be represented as a sequence of pairs - an address, a comma, a value, and a # - following the last # used to represent the registers. The method we employ to simulate 1 step of R will depend on the kind of step.

To simulate a load, we compare the value in the source register with the addresses in the simulated memory cells. Suppose R contains l registers and nonzero memory cells, and the largest size of a value in a register or memory cell or nonzero memory cell address is m. To compare a register with the ith memory address takes time $O(lm^2)$, and so to find the correct memory cell takes time $O(l^2m^2)$. To copy it to the destination register may require pushing the contents of the tape after the destination register to the right by as much as m so that the data can fit in the register. This takes time $O(lm^2 + l^2m) \subseteq O(l^2m^2)$.

To simulate a store is similar. All of the other operations can be carried out at least as fast.

The instruction set was chosen carefully so that at each time step, m, l can each increase by at most 1. (notice that multiplication would have allowed m to double in one step) So at each of the first t steps, $m \leq O(n+t)$ and $l \leq O(t)$. So T can simulate t steps of R in time $O(t^3(n+t)^2)$.

1.2 Simulation of a 1-tape TM

We can simulate t steps of a 1-tape TM T on a RAM machine R in $O(t)$ time, and only using the registers - no extra memory cells are required. To do this, we represent the tape of T that is to the left of the tape head in R_2 and the tape of T that is at and to the right of the tape head in R_1, but with the contents in reverse order. (i.e. left most symbol in the least-significant bits) The reason for this is that R can access the low-order bits of a register in $O(1)$ time, but not the high-order bits, and we will need quick access to the bits near the simulated tape head. The state of T is represented by the program counter of R.

Initially the encoding of the input x will be in R_1, and R_2 contains 0. To simulate 1 step of T, we read the low $O(1)$ bits representing the first symbol in R_1 by using the bit shifting operations and simulate a transition by conditional jumps. We also write the symbol that T would write by using addition and then move the tape head left or right by using more bit shifting and addition. All of this takes $O(1)$ time. To simulate t such steps then takes $O(t)$ time.

2 Circuit model

There are many circuit models, we will describe one. A *circuit* C is a dag (directed, acyclic graph) with ordered children and with each non-minimal node v labeled with a *gate* (function) of arity equal to the indegree of v and each minimal node labeled with an index (an integer representing the name of an input) and each maximal node labeled with an index (representing the name of an output). A Boolean circuit allows only Boolean functions as gates. Not all circuits are Boolean, e.g. perceptrons, arithmetic circuits. The indegree of a node is called its *fanin*.

In this course, if it is not said otherwise, we will use 'circuit' to mean Boolean circuit of maximum fanin 2 and with a single output node. We may occasionally restrict the gates as well, e.g. we may only allow AND, OR, NOT. If we restrict ourselves to commutative gates, then we may ignore the ordering of the children of each node. We can represent circuits in other equivalent forms, such as by giving a tuple representation of the nodes and their children in some topological ordering.

By assigning values to the inputs, we can recursively assign values to each non-minimal node v, including the output, by evaluating the gate at v at the values of the children of v in the order prescribed by the graph. That this process does not depend on the topological ordering in which we choose to evaluate the nodes is the recursion theorem from mathematical logic.

This induces a function from a number of bits equal to the number of input nodes to 1 bit, and so we have a model of computation. But each circuit has only a fixed number of inputs and so only represents a finite function. We can get around this by talking about a circuit *family* $\mathcal{C} = (C_0, C_1, C_2, \dots)$ where each C_n is a circuit on n inputs. Given an encoding function $f : \Sigma \to \{0,1\}^k$, and letting $f^*(x_1 \cdots x_n) = f(x_1) \cdots f(x_n)$ be its homomorphic extension to Σ^*, the language of $\mathcal{C}$ is then

$$\mathcal{L}(\mathcal{C}) = \{x \in \Sigma^* \mid C_{|f^*(x)|}(f^*(x)) = 1\}.$$

It may be easier to simply assume that $\Sigma = \{0,1\}$ and that f is the identity, so that $\mathcal{L}(\mathcal{C}) = \{x \in \Sigma^* \mid C_{|x|}(x) = 1\}$.

But circuit families can compute things that Turing machines cannot. E.g. let L be any language whatsoever, $\langle n \rangle$ be some reasonable encoding of integers into strings, and let C_n output

$$\begin{cases} 1 & \text{if } \langle n \rangle \in L \\ 0 & \text{else} \end{cases}.$$

Each such C_n is just a single node outputting a constant. Then the family $\mathcal{C} = (C_0, C_1, C_2, \dots)$ is certainly not computable (neither the ith bit of the description of the circuit family given i, nor the value C_n would output given an input of size n) if L is not. Such a model of computation is called *nonuniform*, which is meant to suggest that there may not be a single finite object that describes the whole family of circuits. (the circuits in the family cannot be generated in a uniform manner)

A common way to limit the power of circuits is to enforce a uniformity condition: say a circuit family $\mathcal{C} = (C_0, C_1, C_2, \dots)$ is *p-uniform* iff there is a polynomial-time TM T that outputs C_n on input 1^n.

We claim that the languages decided by p-uniform circuit families are exactly the languages decided by polynomial time TMs. To see this, suppose L is decided by a p-uniform circuit family generated by the polynomial time TM T. Then we can combine T with an algorithm that evaluates the circuit to create a polynomial time TM for L. Going the other way, suppose L is decided by a polynomial time TM T. Then given an input 1^n, we can construct in polynomial time a circuit C_n that computes as T does by using the Cook tableau construction (see pg 255 of Sipser). Details are left to the reader.

It should be noted that restricting the fanin to 2 was not so special. p-uniform circuits with fanin of $k \geq 2$ are polynomially equivalent to those with fanin of 2, since one can simply replace a k fanin gate with $O(1)$ fanin 2 gates.

3 Conclusion

The languages that can be decided with only a polynomial amount of resources (time to decide, or time to generate a circuit that decides in our models) is a very robust notion, being equivalent in many different models. If we let $P_{\mathcal{M}}$ be those languages decided using a polynomial amount of resources in model $\mathcal{M}$, then our results so far are

$$\begin{aligned} P_{\text{RAM}} &= P_{\text{1-tape TM}} = P_{k\text{-tape TM}} \\ &= P_{p\text{-uniform circuit, fanin 2}} = P_{p\text{-uniform circuit, fanin }k} \neq P_{\text{circuit}}, \end{aligned}$$

and we simply write P for any of the first 5 quantities.