A New Approach to the First Theory Course

John E. Savage
Department of Computer Science
Brown University

November 2, 1998

Abstract

This article describes a novel first theory course, entitled "Models of Computation," that I've been teaching at Brown University for the last three years with considerable success. In fact, many students tell me that they enjoy it very much, a somewhat unusual reaction for a required theory course. I believe its appeal is due to my efforts to make the material relevant to their experience and interests by introducing new topics, namely, circuits, computer organization, and programming, and covering the standard material in an unusual order. Highlights of the course are that students acquire a good understanding of **NP**-complete languages by midsemester, learn almost all of the material normally taught in a theory course on formal languages and automata and computability, and develop a very good appreciation for the importance of reductions in theoretical (and non-theoretical) computer science. More information on the course can be found at www.cs.brown.edu/courses/cs051.

A New Approach is Needed

As theoreticians, we are convinced that the many beautiful results of our field are very important. As faculty members, we believe that our students should share our enthusiasm for this material. Unfortunately, this is frequently not the case; too many of our students are more interested in building systems than they are in developing language models and exploring the fundamental limits on computation. As a consequence, the experience of teaching a traditional first theory course to students who must take it often leaves both the student and instructor disappointed.

This situation is bad for several reasons. First, as computer systems become ever more complex, modeling and analysis are needed to explore their semantics and performance. Second, the disfavor in which theory finds itself in the undergraduate curriculum today discourages interest in theory by students and results in a loss of administrative support for a subject whose importance is greater today than it has ever been. This situation needs to be rectified.

If students are not going to come to theory willingly, then we need to bring theory to the students. By this I mean that we need to demonstrate its value to them. This means that the material needs to be supplemented, reorganized, and better motivated, and also needs to be grounded in students' experience. Several steps can be taken to make this happen. We can start by not using the word "theory" to describe the first course. I call my course "Models of Computation" to send a strong signal that computer science has many models for different aspects of computation.

Topics Covered in "Models of Computation"

My new course introduces students to the following models: the Boolean function, Boolean formula, straight-line program (SLP), circuit (graph of an SLP), finite-state machine (deterministic and nondeterministic), random-access machine (RAM, a model for the PC), pushdown automaton (PDA), Turing machine, and regular, context-free, and phrase-structure languages. In two lectures I also expose students to two pebbling games used to study space-time tradeoffs and I/O complexity. The four (of nine) assignments done in Scheme serve to cement students' understanding of equivalences between models of computation and to prepare them for two culture lectures on lambda calculus given at the end of the semester.

Models derive their value from the analysis done with them. The new course employs qualitative and quantitative forms of analysis. An example of quantitative analysis is the development of a relationship between the size of a circuit simulating a memory-based machine and the circuit complexity of a function computed by this

machine, thereby exhibiting space-time product lower bounds for machines such as the RAM and Turing machines. Qualitative analysis is illustrated by the simulation of one computation by another and the reduction of one function to another. The latter are used in pumping lemmas and to identify **NP**-complete languages and demonstrate that some problems are unsolvable.

Course Outline

Below we outline the technical content of "Models of Computation," a course of 36 fifty-minute lectures taught over a thirteen-week period. After an introductory lecture, students are given a two-lecture introduction to Scheme that provides enough information for them to do the four Scheme assignments. Students are also given five written assignments and two exams. After a one-lecture review of the mathematical background they are expected to bring to the course, the technical content is presented as outlined below.

Logic Circuits

This course begins with logic circuits and computer organization and by the middle of the semester has covered **NP**-complete languages. This is possible because of the simplicity of circuits. To relate circuits to the experience of programmers, circuits are described as graphs of straight-line programs. Because we later simulate memory-based machines by circuits, it is important that students feel comfortable with them before proceeding.

Since I assume that students have not taken a course on computer organization, I give five lectures on logic circuits to develop their intuition (this number can be decreased if they have the proper background). These lectures cover circuits, formulas, Boolean functions, encoders, decoders, multiplixers, demultiplexers, adders, and multipliers. During these lectures reductions between Boolean functions are introduced, a concept that plays a central role throughout the course. Students are given one Scheme assignment to evaluate an SLP that reinforces the connection between circuits and their programming experience.

Machines with Memory

The deterministic finite-state machine (FSM) model is then introduced and its realization as a sequential circuit is briefly discussed. This sets the stage to construct a logic circuit that simulates a T-step computation by an FSM. From this simulation we produce a computational inequality capturing the idea that this logic circuit is no smaller than the smallest circuit for the function f computed in T steps by the FSM. This simulation concept is reinforced by having students write a Scheme program that generates an SLP simulating a T-step FSM computation. This Scheme program uses T copies of the SLP for the FSM next-state/output function after relabeling steps of the SLP.

The nondeterministic FSM (NFSM) is then introduced to set the stage for the simulation of a nondeterministic Turing machine by a circuit and the identification of a first **NP**-complete problem. Nondeterminism is explained through the unusual device of the "choice agent." The NFSM is defined to be a deterministic FSM with two inputs, the user input and the input provided by the choice agent. When the two inputs are known, the NFSM is deterministic, an idea students now understand. The choice agent must help the user to accept the user input but cannot force the NFSM to accept a string for which the NFSM is not designed. Language recognition is explained in this context and nondeterminism is characterized as a way to describe languages. Once nondeterminism is understood, students can then understand the simulation of nondeterministic machines by circuits.

The RAM is then described as two FSMs running in synchrony, a CPU and a RAM memory unit, whose initial state is determined by a user program. (Here again, the goal is to motivate the student.) We argue that the circuit size of the next-state/output function of the CPU is small by comparison with that of the RAM memory and design a circuit for the latter using circuits encountered at the beginning of the course. From this construction we learn that we can simulate a T-step RAM computation by a circuit of size $O(ST)$, where S is the storage capacity in bits of the RAM memory unit. This leads to the following inequality where f is the function computed by this machine with S bits of memory in T steps and $C_\Omega(f)$ is the size of a smallest circuit over the basis Ω for f:

$$C_\Omega(f) = O(ST)$$

This inequality demonstrates the importance of circuit size as a measure of the complexity of a problem, gives meaning to the space-time product for a RAM computation, and relates this product to circuit size, a fundamental complexity measure. It also demonstrates the importance of circuit simulation.

The deterministic Turing machine (TM) is then described and students are told how it can be used to simulate a RAM and vice versa. This is done to show that one can imagine programming a TM by programming a RAM, the standard mental model, and then translating the result.

Before turning to **NP**-complete languages, we introduce the nondeterministic TM using a NFSM as a control unit. The material of this section is covered in five lectures.

NP-Complete Languages

Language acceptance is defined for the deterministic and nondeterministic Turing machines and the classes of languages **P** and **NP** are introduced.

The next step is to demonstrate that the language-recognition problem for an arbitrary language L in **NP** (it is recognized by an NTM M in $p(n)$ steps on strings of length n in L, $p(n)$ a polynomial in n) can be reduced to testing a circuit for membership in the language CIRCUIT SATISFIABILITY, another use of reductions. Given M and a string $\boldsymbol{w} \in L$ this is done by constructing a circuit in deterministic polynomial time in the length of $\boldsymbol{w}$ that simulates the acceptance by M of $\boldsymbol{w}$ in $p(|\boldsymbol{w}|)$ steps. But this is just a simple extension of the simulation of an NFSM (the number of tape cells used by the TM M on $\boldsymbol{w}$ is bounded) by a circuit, the task of the second and third Scheme assignments. After giving the definition of **NP**-complete languages, we conclude that CIRCUIT SATISFIABILITY is **NP**-complete.

This material is covered in two lectures. Thus, in a total of seven lectures we build from machines with memory to the **NP**-complete languages.

Space-Time Tradeoffs and I/O Complexity

One lecture each is given on space-time tradeoffs and I/O complexity. The former topic is modeled by the placement and movement of pebbles on the vertices of a directed acyclic graph (students readily agree to consider computations of this kind), a fun topic that is readily accessible to students without background. A pebble on a vertex corresponds to the placement of a value in a memory location. A pebble can be placed on a vertex only if it is either an input vertex or all of its predecessors carry pebbles. A pebble can be removed from a vertex at any time. The goal of the game is to pebble all output vertices.

To motivate this material, I tell students that CPUs today operate with first- and second-level caches as well as RAM chips and that a CPU can be two to five times faster than a second-level cache and more than twenty times faster than a RAM chip. As a consequence, it is very important to use the first-level cache well, the situation modeled by the pebble game.

We then introduce the red-blue pebble game to model the movement of data between first and second level memories. (Hot) red pebbles operate under the above rules. (Cool) blue pebbles can be placed on a vertex carrying a red pebble to model the movement of data from primary to secondary memory. Placing a red pebble on a blue pebbled vertex reverses this process. The goal of this game is to minimize the number of swaps between red and blue pebbles (I/O operations) for a given number of red pebbles (the amount of primary memory).

These two lectures allow me to emphasize again the important role that modeling plays in computer science.

Formal Languages and Automata

My treatment of automata and formal languages is not exceptional in any way except perhaps in the amount of time specifically devoted to it. Over the course of ten lectures I demonstrate the equivalence between deterministic and nondeterministic FSMs, show that languages recognized by FSMs are exactly those defined by regular expressions, develop properties of regular languages including the pumping lemma, define language recognition by PDAs, describe language generation by grammars and the Chomsky hierarchy, do parsing of context-free languages using Valiant's version of the Cocke-Kasami-Younger algorithm, and develop properties of context-free languages.

Although only ten lectures are devoted to these topics, the time spent earlier in the course on deterministic and nondeterministic machines and language recognition prepares the students to move quickly through this material.

Computability

The next set of topics, which occupies three lectures, concerns the limits on computability by Turing machines. First the phrase structure languages are shown to be exactly the languages recognizable by Turing machines. Then we use diagonalization to show that some languages are not recognizable by any Turing machines and, using reductions, show that some languages are recursively enumerable but not recursive.

Reductions Between NP-Complete Problems

We then return to complexity classes and place the classes **P** and **NP** in a broader context. We revisit the definition of **NP**-complete languages and do reductions between them. A fourth and final programming assignment has

students write programs to implement reductions between **NP**-complete languages.

Because reductions are introduced in the discussion of Boolean functions, used to show that CIRCUIT SATISFIABILITY is **NP**-complete, employed to demonstrate that some problems are unsolvable, and exercised in several programming assignments, students come away from this course with a very strong understanding of this important and fundamental concept.

Lambda Calculus

The course finishes with two culture lectures on lambda calculus. The purpose of these lectures is to provide students with a very primitive model of computation that they can appreciate given their extensive experience with Scheme and to give them a taste of the semantics of formal languages. It also allows me to finish the semester by telling them they have seen a very broad range of topics representative of theoretical computer science yet relevant to many of their interests.

The Textbook

This course makes use of my book [1] for this course. It uses a portion of Chapter 2 (Logic Circuits), all of Chapter 3 (Machines with Memory), Chapter 4 (Finite-State Machines and Pushdown Automata), and Chapter 5 (Computability). It also uses a small portion of Chapter 8 (Complexity Classes) on **NP**-complete languages as well as the introductory sections of Chapter 10 (Space-Time Tradeoffs) and Chapter 11 (Memory Hierarchy Tradeoffs). Information on the book can be found at www.cs.brown/people/jes/book/book.html.

Bibliography

[1] John E. Savage, **Models of Computation**: *Exploring the Power of Computing*, Addison Wesley, Reading, MA, 1998.