1 Basic Resources, Turing Machines, Circuits

CS 6810 – Theory of Computing, Fall 2012
Instructor: David Steurer
Scribe: Sujay Jayakar (dsj36)
Date: 08/23/2012

1.1 What is Complexity Theory?

The heart of complexity theory lies in *problems*, *resources*, and studying the interaction between them when we impose constraints. In other words, how do resource constraints affect our ability to solve problems? There are two facets to this question: First, we can devise algorithms to get upper bounds on the resources needed to solve a problem, or we could prove impossibility results to establish a lower bound on the necessary resources.

Additionally, complexity theory explores the relationships within problems and resources. If solving one problem A amounts to solving another problem B, we may say that A is easier than B or A *reduces* to B. In a similar nature, one class A of resources may be more powerful than another B if computation in A can *simulate* computation in B.

1.1.1 Problems

Problems can take many forms, but the one we will study most is the *decision problem*. Formally, a decision problem is a subset (or *language*) $L \subseteq \{0,1\}^*$, and solving the problem amounts to telling whether a string $x \in \{0,1\}^*$ is in L. One common example of a decision problem is 3SAT, where L is the set of satisfiable 3CNF formulas.

A more general class of problems are *search problems*. Intuitively, given an input $x \in \{0,1\}^*$, there is a set of acceptable outputs $A(x)$. This problem, then, is a relation on $\{0,1\}^*$, where

$$R = \{(x, y) : y \text{ is acceptable for } x\}.$$

Given an input x, solving the problem involves finding a y such that $(x, y) \in R$.

A middle ground between the two previous varieties of problems are *promise problems*. The set $\{0,1\}^*$ has two disjoint subsets L_{YES} and L_{NO}. On input x, a solution must input 1 if $x \in L_{YES}$, 0 if $x \in L_{NO}$, and anything if x is in neither. The term *promise* comes from the observation that the problem is equivalent to a decision problem when the algorithm is promised that the input lives in $L_{YES} \cup L_{NO}$.

1.1.2 Resources

For each possible type of computational model, there are a variety of resources that we would like to measure and restrict. Here, we outline common models and their measures.

- Turing machines: time (number of operations), space (maximum number of tape cells used), randomness, advice
- Circuits: size (number of vertices), depth (longest path through circuit)
- Protocols: communication cost
- Resolution proofs: length
- Linear programs: number of constraints (number of facets of polytope domain)

1.1.3 Reductions

In general, we say that a problem A reduces to a problem B if we can solve problem A using some knowledge of how to solve problem B. We can formalize this notion in several ways.

- Turing/Cook reductions: Solve problem A using a solution to problem B as a subroutine.
- Karp reductions: Map instances of the decision problem A into instances of decision problem B such that answers are preserved. (This also makes sense for promise problems.)
- Levin reductions: Map instances of A into B such that solutions of B can be efficiently pulled back into A. (This makes sense for search problems.)

The existence of a reduction from A to B establishes that B is hard if A is hard and that A is easy if B is easy. The first approach is used for NP-hard reductions, and the second is used for linear programming relaxations, where the rounding algorithm is the component of the Levin reduction that recovers solutions to the original problem.

1.2 Turing Machines

When coming up with a model of computation, we would like to restrict the description of functions from inputs x to outputs y. Turing observed that two restrictions yielded a useful formalism: The functions must be finitely describable, and the function must be decomposable into "local" steps.

1.2.1 Description

We will begin describing Turing machines by outlining their state at a given point of time, or their *configuration*. Turing machines have k semi-infinite tapes, each blank except for finitely many cells. Each nonempty cell contains a symbol from a finite alphabet Σ, so we may view the state of the tapes as an element of $(\Sigma^*)^k$. Each tape has a head that reads a particular cell, and the head may only be in a single position at each point in time. Therefore, the state of the heads may be described by an element of $\mathbb{N}^k$. In addition, we fix a finite set of states Q, and the machine is in any single state q at any given time.

The remainder of a Turing machine's description is the transition function from one configuration to another. Here, we restrict the function to be local. That is, it may only depend on the cell under each of the k tape heads and the current state q. Furthermore, the tape heads may only write to the current cell and move by at most one cell in either direction. This locality directly guarantees us a finite description.

Input can be done by placing the input string on one of the tapes, and the machine can output a result by writing it on a particular tape and entering a distinguished halt state. Note that Turing machines need not halt. For a Turing machine M, set $M(x)$ to be the machine's output y if it halts and $\perp$ if it does not.

1.2.2 Complexity

We can relate Turing machines to decision problems by saying that a machine M solves L if $M(x) = 1$ if and only if $x \in L$. We say that M is t-timebounded if for all inputs x, M halts in time $t(|x|)$, where $|x|$ is the length of x. Space bounds are defined in a similar way.

Given a function $t : \mathbb{N} \longrightarrow \mathbb{N}$, we can define the class $TIME(t)$ to be the set of all languages solved by a t-timebounded Turing machine. Our familiar class of "efficient" polynomial time computation is then $P = \bigcup_{c \geqslant 1} TIME(n^c)$.

1.2.3 Universal Computation

Since a Turing machine's description is finite, we can write it down as a bitstring. Therefore, it is a natural question to consider whether there is a Turing machine that takes another's description M as input, simulates M, and outputs its result. In the case where M uses no more than k tapes, simulation is easy. We simply use $k + 2$ tapes, k of them for M's tapes, one for M's description, and the final one for its state.

If we would like to simulate Turing machines with arbitrarily many tapes, simulation is more involved. Instead of using many tapes, we can store the configuration of the machine on a single tape and sweep through it, remembering the symbols at the heads. After our first sweep, we decide what changes to make and make another sweep to alter the state. The drawback of this approach is that

it requires two passes per step of M, simulating T steps in T^2 time. A clever trick can bring this slowdown to $T \log T$ – see section 1.7 in the textbook.

1.3 Circuits

Circuits are an alternate model of computation. In general, a circuit is a labeled directed acyclic graph where every vertex has indegree less than or equal to two. In addition, there are three types of vertex labels.

- Sources: Indegree 0, labeled with input variable
- Sinks: Outdegree 0, labeled with output variable
- Gates: Labeled with Boolean operator ($\wedge$, $\vee$, or $\neg$)

If a circuit C has sinks $x = x_1 \dots x_n$ and sinks $y = y_1 \dots y_m$, we set $C(x) = y$, where y is the result of evaluating C on input x. To handle inputs of arbitrary size, we may consider a sequence of circuits $\{C_n\}$, where $C(x)$ is defined to be $C_{|x|}$.

1.3.1 Resource Bounds

Define the size of a circuit to be its number of vertices. Then given a function s, consider the class of problems $SIZE(s)$ to be the languages L computed by circuits of size no more than $s(n)$. As before, define $PSIZE$ to be $\bigcup_{c \geqslant 1} SIZE(n^c)$.

Theorem 1.1. *There exists $f : \{0,1\}^n \longrightarrow \{0,1\}$ such that $f \notin SIZE(2^n/n)$.*

Proof. We can show the existence of f by a counting argument of sorts. The number of all functions from $\{0,1\}^n$ to $\{0,1\}$ is 2^{2^n}. We can upper bound the number of circuits of size s by considering how many bits it would take to describe them. Each node must have at most two parent indices, each taking $\log s$ bits, and an annotation in $\{\neg, \wedge, \vee, \text{input/output}\}$, yielding two more bits. Therefore, the number of circuits is at most $2^{s(2 \log s + 2)}$, so to represent every function, we need a size of at least s^*, where $s^*(2 \log s^* + 2) = 2^n$. Solving for s^* gives us s^* roughly equal to $2^n/2n$. □