Algorithms and Church’s Thesis

Sipser  pages 154 - 163
Enumeration

• Recall we said that acceptance by a TM was also called recursively enumerable.
• An enumerator is a machine that “enumerates” all strings in a language.
• Think of it as a Turing machine with a printer.
  – Every string is eventually “printed”
  – Some strings are “printed more than once”
Computable Functions

• Importance of having precise definitions of *effectively* computable functions, or algorithms, was understood in the 1930's. There were several attempts to formalize the basic notions of computability:
  – Turing Machines (1936)
  – Post Systems (1936)
  – Recursive Functions (Kleene, 1936)
  – Markov Algorithms (1947)
  – $\lambda$-calculus (Church 1936)

• On the surface, these approaches look quite different. It turned out, however, that they are all equivalent! All these, and all later formalizations (combinatory logic, *while* programs, C programs, etc.) give essentially the same meaning to the word *algorithm*.
Church’s Thesis

• The statement that these formalizations correspond to the intuitive concept of computability is known as *Church's Thesis*.

• Church's Thesis is a belief, not a theorem.

• (though we often act as if we believe it is true, even though we don’t know its is true)
Power of Turing Machines (1)

• Recall the Church Thesis: *Every problem that has an algorithmic solution can be solved by a Turing Machine*!

• How do we become convinced that it is reasonable to believe this thesis?

• **First**, we can develop some programming techniques for TM's, allowing us to write machines for more and more complicated problems. Structuring states and tape symbols is particularly useful. Then, there is a possibility to use one TM as a subroutine for another. After having written enough TM's, we may get a feeling that everything that we can program in a convenient programming language could be done with TM.
Power of Turing Machines (2)

- Second, we can consider some generalizations of the concept of TM (multitape TM's, non-deterministic TM's, ...) and prove that they are essentially just as powerful as the plain TM's.

- Finally, we can prove that all proposed formalizations of the concept of *computable*, of which TM's is only one, are equivalent. In later lectures we will look at both Kleene and Church’s systems.
Computation using Numerical Functions

• We’re used to thinking about computation as something we do with numbers (e.g. on the naturals)

• What kinds of functions from numbers to numbers can we actually compute?

• To study this, we make a very careful selection of building blocks
Turing-computable functions

• To formalize the connection between partial recursive functions and Turing machines, we need to describe how to use TM’s to compute functions on $\mathbb{N}$.

• We say a function $f : \mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$ is Turing-computable if there exists a TM that, when started in configuration $q_0 1^{n_1} \square 1^{n_2} \square \ldots \square 1^{n_k}$, halts with just $1^f(n_1,n_2,\ldots,n_k)$ on the tape.

• Fact: $f$ is Turing-computable iff it is partial recursive.