Computability and Incompleteness Fact Sheets

Computability

Definition. A *Turing machine* is given by:

- A finite set of *symbols*, $s_1, \ldots, s_m$ (including a "blank" symbol)
- A finite set of *states*, $q_1, \ldots, q_n$ (including a special "start" state)
- A finite set of *instructions*, each of the form

  > If in state q_i scanning symbol s_j, perform act A and go to state q_k

  where A is either "move right," "move left," or "write symbol s_l."

The notion of a "computation" of a Turing machine can be described in terms of the data above.

From now on, when I write "let f be a function from strings to strings," I mean that there is a finite set of symbols Σ such that f is a function from strings of symbols in Σ to strings of symbols in Σ. I will also adopt the analogous convention for sets.

Definition. Let f be a function from strings to strings. Then f is *computable* (or *recursive*) if there is a Turing machine M that works as follows: when M is started with its input head at the beginning of the string x (on an otherwise blank tape), it eventually halts with its head at the beginning of the string $f(x)$.

Definition. Let S be a set of strings. Then S is *computable* (or *decidable*, or *recursive*) if there is a Turing machine M that works as follows: when M is started with its input head at the beginning of the string x, then

- if x is in S, then M eventually halts, with its head on a special "yes" symbol; and
- if x is not in S, then M eventually halts, with its head on a special "no" symbol.

The Church-Turing Thesis. A function is *computable* in the intuitive sense if and only if it is *computable* according to the definition above; and similarly for sets.

Unsolvable problems

Definition. Let S be a set of strings. Then S is *computably enumerable* (or *semi-decidable* or *recursively enumerable*) if there is a Turing machine M that works as follows: when M is started with its input head at the beginning of the string x, then:

- If x is in S, the machine halts with its head on a special "yes" symbol; and
- If x is not in S, the machine never halts.

It should be clear that any computable set is computably enumerable. (Why?) The following shows that there are computably enumerable sets that are not computable.

Definition. Suppose we fix a reasonable encoding of Turing machines as strings of symbols. Let K be the set of encodings of pairs $\langle M, x\rangle$ such that Turing machine M halts on input x. The problem of deciding whether or not a string is in K is known as the *halting problem*; it amounts to answering the question "does machine M halt on input x?

Theorem (Turing). K is computably enumerable, but not computable.

Turing showed that K is computably enumerable by first showing that there is a "universal Turing machine," which can simulate the behavior of any machine that is specified as input. A machine can then decide K as follows: on input $\langle M, x\rangle$, use the universal Turing machine to simulate the behavior of M on input x, and return "yes" if the simulation eventually halts.

Let us sketch a proof that K is not computable. Suppose otherwise; that is, suppose K is computed by a machine M. Then we could build a machine N that works as follows. On input x:

1. N calls M with the question, "does machine x halt on input x"?
2. If the answer is "yes," N goes into an infinite loop. If the answer is "no," N halts.

Now ask, does N halt when you input an encoding of N itself? First N calls M with the question "does N halt on input N?" But if M says "yes," N doesn't halt; and if M says "no," N halts. This contradicts the assumption that M decides the question correctly. So there is no such machine M.

It turns out that undecidable problems can arise in very natural contexts. Here are two more examples:

1. Consider predicate logic with at least one binary relation symbol. The question, "is formula F provable from the axioms and rules of predicate logic?" is computably enumerable but not computable.

2. "Peano arithmetic" is a simple set of axioms for arithmetic. The question, "is formula F provable from the axioms of arithmetic?" is computably enumerable but not computable.

Incompleteness

Consider strings of symbols that represent formulas in predicate logic.

Definition. Let A be a set of axioms.

- A is *complete* if for every sentence F in the language, either one can prove F from A, or one can prove $\sim F$ from A.

- A is *consistent* if there is no sentence F such that one can prove both F and $\sim F$ from A.

Here are some natural questions:

1. Is there a complete, computable set of axioms for mathematics?

2. Given a set of axioms for mathematics, can one prove that the axioms are consistent (using only "minimal" assumptions)?

3. Given a mathematical statement, is there an algorithm that decides whether or not the statement is true, or provable from a given set of axioms?

Hilbert proposed representing mathematics with formal languages and formal proof systems, for the purpose of addressing these questions.

Gödel's first incompleteness theorem. There is no complete, consistent, computable set of axioms strong enough to prove a certain collection of basic facts of arithmetic.

Gödel's second incompleteness theorem. No consistent, computable set of axioms strong enough to prove a certain collection of basic facts of arithmetic can prove its own consistency.

Theorem. There is no algorithm that decides whether or not a statement of arithmetic is true. Given a "reasonable" set of axioms for arithmetic, there is no algorithm that decides whether or not a formula is provable from the axioms.