Sapienza Università di Roma
Dipartimento di Informatica

FOUNDATIONS OF COMPUTER SCIENCE
LECTURE 11: Church-Turing thesis

Prof. Daniele Gorla

Another variant of TM: Enumerators

Def.: An ***enumerator*** is a 2-tapes (called *working tape* and *print tape*) Turing machine

$$E = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{print}, q_{reject} \rangle \text{ where}$$

- Q is a finite, non-empty set of *states*.
- Σ is a finite, non-empty set of the *output / print alphabet*
- Γ is a finite, non-empty set of the *working tape alphabet*
- δ is the transition function. $\delta : Q \times \Gamma \to Q \times \Gamma \times \Sigma^* \times \{L, R\}$
- $q_0 \in Q$ is the start state
- $q_{print} \in Q$ is the print state.
- q_{reject} is the reject state with $q_{print} \neq q_{reject}$ Theoretically it never stops

Intuitively, an enumerator E can use the print tape as an output device to print strings

- It starts with a blank input on its working tape
- If it doesn't halt, it may print an infinite list of strings
- Printing corresponds to entering the q_{print} state and the content of the 2nd tape is printed

The ***language enumerated by E*** is the collection of all the strings that it eventually prints

→ E may generate the strings of the language in any order, possibly with repetitions

The set of languages enumerated by some enumerator is called ***recursively enumerable***.

Enumerators are equivalent to TMs

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Thm.: Every language enumerated by some enumerator is accepted by a TM, and vice versa.

Proof

➔ Let E be an enumerator that enumerates language L.

The TM M that recognizes L has 3 tapes and works in the following way:

On input w (placed on the 1st tape):

Run E by using the 2nd and 3rd tape as working and print tape, respectively;

Every time that E prints a string, compare it with w;

If they coincide, ACCEPT.

Clearly, M accepts all and only those strings that appear on E's list.

🡐 If a TM M accepts a language L, we can construct the following enumerator E for L.

Say that $s_1, s_2, s_3,\ldots$ is a list of all possible strings in Σ^*. E behaves in this way:

```
forall i = 1,2,3,... do
    for j = 1 to i do
        run M for at most i steps on input s_j
        if accepts, print s_j
```

If M accepts s, eventually it will appear on the list generated by E

(actually, it will appear infinitely many times). Q.E.D.

Models of computation

- We have presented several variants of the TM model, all equivalent in power.
- Many other models of general purpose computation have been proposed.
- Some of these models are very similar to TMs, but others are quite different.
- All share the essential feature of having unrestricted access to unlimited memory
- Remarkably, *all* models with that feature turn out to be equivalent in power, provided that they satisfy reasonable requirements (e.g., perform only a finite amount of work in one single step)
- Something similar happens to programming languages
  - Many of them look quite different from one another in style and structure.
  - Can some algorithm be programmed in one of them and not in the others?
  - No: we can (more or less easily) compile one into another
  - This means that the two languages describe *exactly* the same class of algorithms
- This equivalence phenomenon has an important philosophical (and practical) corollary: Even though we can imagine many different computational models, the class of algorithms that they describe remains the same.
- This phenomenon has had profound implications for mathematics!

What is an algorithm?

- Informally, an *algorithm* is a collection of simple instructions for solving some task
- Algorithms play an important role in mathematics:
  - Ancient mathematical literature contains descriptions of algorithms for a variety of tasks (e.g., prime numbers with the Sieve of Eratosthenes, or greatest common divisor with Euclid's method)
  - In contemporary mathematics, algorithms abound (proofs by construction usually provide a way to effectively build a solution of a given problem)
- The notion of algorithm itself was not defined precisely until the 20th century
- Before that, mathematicians had an intuitive notion of what algorithms were

  → the intuitive notion was insufficient for deeply understanding algorithms
- The formal definition of *algorithm* came in the 1930's
  - Alonzo Church (λ*-calculus*, a notational system to formalize computations by function application)
  - Alan Turing (with his *machines*)
  - Kurt Gödel (*general recursive functions*: the smallest class of partial functions that is closed under composition, recursion, and minimization, and includes zero, successor, and projections)
- These definitions were shown to be equivalent.
- This connection between the informal notion of algorithm and the precise definition has been later on called the ***Church–Turing thesis***.

The Church-Turing thesis

It can be formulated as follows:

«A function on the natural numbers can be calculated by an effective method

if and only if it is computable by a Turing machine»

or, more compactly:

«Every effectively calculable function is computable»

In all formulations, there is a term («effective»/«effectively»/…) that should be ‘‘defined’’.

One possibility is:

«An Effective Method is a method each step of which is precisely predetermined

and which is certain to produce the answer in a finite number of steps»

The two key ingredients here are:

1. A precise sequence of (elementary) steps
2. The answer must be produced in finite time (always terminates)

This is the intuitive notion of algorithm that mathematicians had for centuries!

Hence, the Church-Turing thesis can be also stated as:

«Every intuitive algorithm can be implemented by a Turing Machine»

Algorithms as Deciders (1)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

From now on, we'll speak about TMs, but our real focus will be on algorithms (i.e., TMs merely serve as a precise model for defining algorithms)

→ since algorithms always terminate, our focus will (mostly) be on ***deciders***

Moreover, we shall describe TMs at a higher abtraction level

Up to now, we have described a TM via

- its *formal description* (that fully spells out the states, transition function, and so on)

  → this is the lowest, most detailed level of description (almost never used)

- its *implementation description* (by using English prose to describe the way in which the TM moves its head and it stores data on its tape)

  → we do not give details on states nor on transitions

From now on, we shall describe a TM via

- its *high-level description* (by using English prose to describe the underlying algorithm, ignoring all the implementation details)

Algorithms as Deciders (2)

Algorithms usually take arbitrary inputs (e.g., arrays, graphs, grammars, automata, …) and return outputs (from the simple YES/NO, to more complex outputs)

For outputs: we can consider multitape TMs, with a reserved tape for the output

(similar to the enumerator seen a few slides before)

For input: TMs only have one single string as input

→ need for encoding more complex inputs (and possibly outputs)

- Strings can easily represent polynomials, graphs, grammars, automata, and any combination of those objects.

  → the encoding of an object O is a string is written $\langle O \rangle$

  → with several objects $O_1,\ldots,O_k$, their encoding into one single string is $\langle O_1,\ldots,O_k \rangle$
- The encoding can be done in many reasonable ways (it doesn't matter which one we pick because a TM can always translate one such encoding into another)
- If the input description is supposed to be the encoding of an object, the TM first implicitly tests whether the input properly encodes an object of the desired form: if so, it decodes the representation before elaborating it; otherwise, it rejects. a sort of type-checking on the input

Problems as Languages

- Algorithms are procedures for solving problems
- Our aim will now be to see if some problems admit an algorithm (i.e., a decider) or not
- For the sake of uniformity, we shall see all the solutions of a problem as a language
- This is not restricting: indeed, given a problem P, we can always define a proper encoding $\langle - \rangle$ for the instances of P and have that

$$p \text{ is a solution for P} \quad \text{iff} \quad \langle p \rangle \in L(\text{P})$$

where $L(\text{P})$ is defined to be $\{\langle p \rangle \mid p \text{ is a solution of P}\}$

EXAMPLE: P = «does the string w belong to language L?»

- This problem has 2 inputs: w and L
- Since L is a language, there exists a grammar G_L that generates it
- A grammar (of whatever type) is a 4-tuple of finite elements
- We can encode all grammars by numbering terminals and variables, and by accordingly rewriting their rules
- So, we can formulate P as the language $L(\text{P}) = \{\langle w,G \rangle \mid w \in L(G)\}$

  and have that $w \in L$ if and only if $\langle w,G_L \rangle \in L(\text{P})$

Decision vs Optimization Problems (1)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Seeing a problem as a language suits well when the problem has a binary answer (YES/NO)

EXAMPLES:
- «does the string w belong to language L?»
- «does graph G contain a path from u to v?»
- «is n a prime number?»
- …

These are called ***decision problem***.

What about ***optimization problems***, i.e. problems where the answer is not YES/NO but they require to compute the best value of some parameter?

EXAMPLES:
- «what is the shortes path from u to v in graph G?»
- «what is the greatest common divisor of n and m?»
- …

We can reformulate optimization problems into decision problems by putting a bound on the parameter to be optimized and by asking whether a solution of that value exists or not.

EXAMPLES:
- «is there a path <u>of length at most k</u> from u to v in graph G?»
- «is there a number <u>greater than or equal to k</u> that divides n and m?»
- …

Decision vs Optimization Problems (2)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Considering the decision problem instead of the corresponding optimization problem is enough if we want to prove that the optimization problem is difficult/unsolvable:

Decision prob. is hard/unsolvable → *Optimization prob. is hard/unsolvable*

Indeed, if the optimization problem is easy/solvable, then there exists a (efficient) solution for the corresponding decision problem.

EXAMPLE: if you want to say whether a path of length k exists in G from u to v (decision problem) and you know how to solve the shortest path problem (optimization problem), then

1. run shortest path from u to v
2. if what you obtain is $\leq k$, then return YES, otherwise return NO

If we're just interested in the existence of a solution, also the converse implication holds.

EXAMPLE: if you want to find the length of the shortest path from u to v in G (optimization problem) and you know how to solve the associated decision problem, then

for all $k \geq 0$: see if G has a path of length k from u to v
if YES, then return k