91.304 Foundations of (Theoretical) Computer Science

Chapter 3 Lecture Notes (Section 3.3: Definition of Algorithm)

David Martin
dm@cs.uml.edu

With modifications by Prof. Karen Daniels, Fall 2012

Overview

- Algorithm
  - Intuitive definition
- Hilbert's Problems
  - Show how definition of algorithm was crucial to one mathematical problem
  - Introduce Church-Turing Thesis
- Terminology for Describing Turing Machines
  - Levels of description

What’s It All About?

- Algorithm:
  - steps for the computer to follow to solve a problem
  - *well-defined computational procedure* that transforms input into output
  - (analysis of algorithms is studied in 91.404)

Hilbert's Problems

- Show how definition of algorithm was crucial to one mathematical problem.
  - Mathematician David Hilbert (in 1900) posed his famous grand-challenge list of 23 problems to the mathematical community.
    - 10^{th} problem: devise a "process" that tests whether a given polynomial has an *integral* root.
      - Root is assignment of values to variables such that result = 0.
        - Example (single variable with integer coefficients):

$$f(x) = x^2 - 4x + 4$$

What are the root(s)? Are they integers?

Hilbert's Problems

- 10^{th} problem asks if D is <u>*decidable*</u>.

$$D = \{ p \mid p \text{ is a polynomial with an integral root} \}$$

  - It is not decidable!
  - It is *Turing recognizable*.
    - Motivate key idea using simpler problem:

$$D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \}$$

      - TM M_1 recognizing D_1:
        - M_1 = "The input is a polynomial p over variable x.
          1. Evaluate p with x set successively to the values 0, 1, -1, 2, -2, … If at any point p evaluates to 0, accept."
        - If an integral root exists, M_1 will find *one* and accept.
        - If no integral root exists, M_1 runs forever…

Hilbert's Problems

- 10^{th} problem asks if D is ***decidable***.

$$D = \{ p \mid p \text{ is a polynomial with an integral root} \}$$

possibly multivariate

  - It is not decidable!
  - It is ***Turing recognizable***.
    - TM M recognizing D:
      - Similar to M_1 but tries all possible settings of variables to integral values.
  - M and M_1 are ***recognizers***, not deciders!
  - M_1 (not M) can be converted to a decider via clever bounds on roots: $\pm k \left(\frac{c_{max}}{c_1} \right)$
      - k = number of terms
      - c_{max} = coefficient with largest absolute value
      - c_1 = coefficient of highest order term
      - Matijasevic's Theorem: such bounds don't exist for M.

The Church-Turing Thesis

- **Any algorithmic-functional procedure that can be done at all can be done by a Turing machine**
- This isn't provable, because “algorithmic-functional procedure” is vague. But this thesis (law) has not been in serious doubt for many decades now.
- TMs are probably the most commonly used *low-level* formalism for functional algorithms and computation
  - Commonly used high-level formalisms include pseudocode and all actual programming languages. By Church-Turing thesis, these are all equivalent in terms of what they can (eventually) do.
  - Of course they have different ease-of-programming and time/memory efficiency characteristics.

Intuitive notion of algorithms “equals” Turing machine algorithms.

Terminology for Describing Turing Machines

- Some ways to describe Turing machine computation:
  - Formal description (7-tuple)
    - $M=(Q,\Sigma,\Gamma,\delta,q_0,q_{acc},q_{rej})$
  - Detailed state diagram.
  - Implementation-level description
    - English prose describing way TM moves its head and modifies its tape.
  - *Instantaneous descriptions* (IDs) specifying snapshots of tape and read-write head position as computation progresses on a specific input.
  - High-level English prose describing *algorithm*.
    - As in M_1 (finding integral roots for polynomial over x)
  - Comfort with one level allows "transition" to less detailed level of description…
  - See next slide for format and notation for high-level description.

We have used these already.

Terminology for Describing Turing Machines (continued)

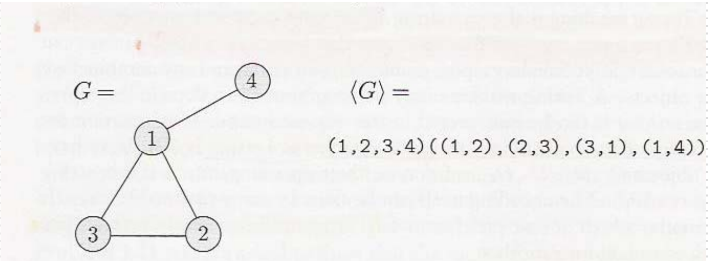
- Input to TM is a string.
  - Encoding an object O as a string: $< O >$
  - Encoding multiple objects as strings:
    - $O_1, O_2, \ldots, O_k$ is encoded as: $< O_1, O_2, \ldots, O_k >$
  - Turing machine can translate one encoding into another, so just pick a reasonable encoding.

Terminology for Describing Turing Machines (continued)

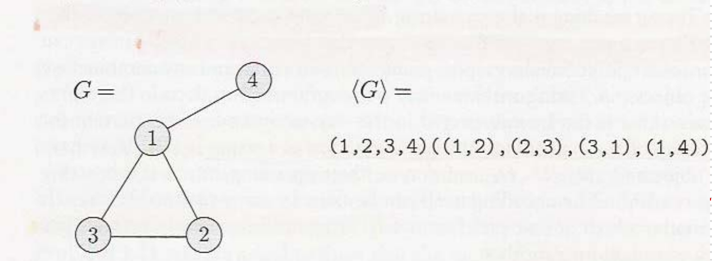
- Example: $A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \}$
- M_3="On input <G>:
  1. Select first node of G and mark it.
  2. Repeat step 3 until no new nodes are marked:
  3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
  4. Scan all nodes of G to check if they are all marked. If so, *accept*; otherwise, *reject*."

Terminology for Describing Turing Machines (continued)

- Practice implementation-level details for M_3:
  - Check if input encoding <*G*> represents a legal instance of a graph.
    - No repetitions in node list.
      - How to check?
    - Each node in edge list also appears in node list.
  - See next slide for detail on steps 1-4.



Terminology for Describing Turing Machines (continued)



- Example: $A = \{ <G> \mid G \text{ is a connected undirected graph} \}$
- M_3 = "On input <G>:
  1. Select first node of G and mark it.
     1. *Dot* leftmost "digit"
  2. Repeat step 3 until no new nodes are marked:
  3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
     1. Find undotted node n_1 (in node list); *underline* it.
     2. Find dotted node n_2 (in node list); *underline* it.
     3. Check if underlined pair (n_1, n_2) appears in edge list.
        1. If so, dot n_1, remove underlines, restart step 2.
        2. Otherwise, check more edge(s).
     4. If (n_1, n_2) does not appear in edge list, try another n_2 .
  4. Scan all nodes of G to check if they are all marked. If so, *accept*; otherwise, *reject*."
     1. Check if all nodes are dotted.