The Definition of Algorithm

An informal discussion

- Informally speaking, an *algorithm* is a collection of simple instructions for carrying out a task

- **Note:** in everyday life algorithms are called *procedures* or *recipes*

- Algorithms play an important role in mathematics: Ancient mathematical literature contains descriptions of algorithms for a variety of tasks:

  - finding prime numbers

  - finding the greatest common divisors

  - . . .

Algorithm: the concept

- The notion of algorithm itself was not defi ned precisely until the twentieth century

- Before 20-th century mathematicians had an intuitive notion of what algorithms were and relied on that notion when using algorithms

- Intuitive notion of algorithm was insuffi cient for gaining deeper understanding of algorithms

- Hilbert's program forced the development of a formal notion of an algorithm

Hilbert's program

- In 1900, David Hilbert delivered an address at the International Congress of Mathematicians in Paris

- In his lecture, Hilbert identifi ed 23 mathematical problems and posed them as a *challenge* for the coming century

- Among these problems, 10-th problem required a "process according to which it can be determined whether a polynomial has an integral root"

Note: Hilbert did not use the term *algorithm*

More on Hilbert's 10-th problem

- A polynomial is a sum of terms, where a term is a product of certain variables and constants called coeffi cients

- **Example:**
    - Terms: $7x^4y^5z^2$, $3x^4$, $4^xy^4z^7u^8$, $27x$, 100
    - Polynomial: $7x^4y^5z^2 + 3x^4 + 4^xy^4z^7u^8 + 27x + 100$

- **Polynomial root:** if $P(x, y, \ldots, z)$ is a polynomial, a root of P is an assignment $x := a, y := b, \ldots, z := c$ to the variables $x, y, \ldots, z$ such that $P(a, b, \ldots, c) = 0$

- A root is integral if $a, b, \ldots, c$ are integers

Observations

- Hilbert's 10-th problem is unsolvable: i.e., there is no algorithm that can decide whether a polynomial $P(x, y, \ldots, z)$ has an integral root.

- The intuitive concept of algorithm was useless for showing that no algorithm exists to solve Hilbert's 10-th problem

- Proving that an algorithm doesn't exist to solve a given problem requires a formal defi nition of algorithm.

Note: proving the existence of an object can be done by identifying one,

Algorithm: formal definition

- Alonzo Church and Alan Turing in 1936 came with formal defi nitions for the concept of algorithm

- Church used a notational system called λ-calculus to defi ne algorithms

- Turing used his "Turing Machines" to defi ne algorithms

- These two defi nitions were shown to be equivalent

Note: this connection between the informal concept of algorithm and the

Algorithm: more formal definitions

Other formal defi nitions of algorithms have been provided by: Kleene using *recursive functions*, Markov using rewriting (derivation) rules with a grammar called *normal algorithms*.

Essentially all these formal concepts of algorithm are equivalent among them and are equivalent with Turing Machines

Solving Hilbert's problem

In 1970 Yuri Matijasevič, building on work of Martin Davis, Hilary Putnam, an Julia Robinson showed that no algorithm exists for testing whether a polynomial has integral roots

Understanding Matijasevič work

Let us rephrase Hilbert's tenth problem using our terminology:

- Consider the language:

  $D = \{P | P$ is a polynomial with an integral root $\}$

- Hilbert's tenth problem asks in essence whether D is decidable

- The answer is negative

A simpler problem

Consider only polynomials in one variable such as $P(x) = 4x^3 - 2x^2 + x - 7$

- Let $D_1 = \{P | P$ is a polynomial over x with integral coefficients$\}$

- A Turing Machine M_1 that recognizes D_1 follows:

  M_1 = "The input is a polynomial P over the variable x

  1. Evaluate P with x set successively to the values 0, 1,-1,2,-2,3,-3,...
     If at any point the polynomial evaluates to 0 accept"

Note, a similar machine, M, can be designed for the language D. However, M and M_1 are recog-

Analyzing M_1

- If P has an integral root, M_1 will eventually find it and accept

- If P has no integral root, M_1 will run forever

Note: the integral roots a of a polynomial in D_1, if they exist, must lie in the interval

$$-k\frac{C_{max}}{C_1} \leq a \leq k\frac{C_{max}}{C_1}$$

where:

1. k is the number of terms in P,

2. C_{max} is the coefficient with the largest absolute value, and

3. C_1 is the coefficient of the highest order term

A TM that decides D_1

We can use the inequality $-k\frac{C_{max}}{C_1} \le a \le k\frac{C_{max}}{C_1}$ to design an algorithm M_1' that decides D_1:

M_1' = "The input is a polynomial P over the variable x

1. Evaluate P with x set successively to the values

$$0, 1, -1, 2, -2, 3, -3, \ldots \in [-k\frac{C_{max}}{C_1}, k\frac{C_{max}}{C_1}]$$

   If at any point the polynomial evaluates to 0 *accept*; otherwise *reject*."

Note: since there are only a finite number of integers in the interval $[-k\frac{C_{max}}{C_1}, k\frac{C_{max}}{C_1}]$, M_1' always terminate. Therefore M_1' is a decider of D_1.

Matijasevič theorem

Matijasevič has shown that it is impossible to calculate such bounds for multi-variable polynomials. Hence, D is undecidable

A turning point

- We continue to speak of Turing machines, but our real focus is on algorithms

- Turing machine merely serves as a precise model for the defi nition of algorithm. Therefore we can skip over extensive theory of Turing machines

- We only need to be comfortable enough with Turing machine to be sure that they capture all algorithms

Note: recently more powerful models of algorithm are developed under

the concept of *hypercomputation*. See journal "Theoretical Computer

Standardizing our model

Question: *what is the right level of detail to give when describing a Turing machine algorithm?*

Note: this is a common question asked especially when preparing solutions to various problems such as exercises and problems given in assignments and exams during the process of learning Theory of Computation

Answer

The three possibilities are:

1. *Formal description:* spells out in full all 7 components of a Turing machine. This is the lowest, most detailed level of description.

2. *Implementation description:* use English prose to describe the way Turing machine moves its head and the way it stores data on its tape. No details of state transitions are given

3. *High-level description:* use English prose to describe the algorithm, ignoring the implementation model. No need to mention how machine manages its head and tape.

Note

- So far we have used both formal description, and implementation description

- Here we set up a format and notation for describing Turing machine while approaching *decidability* or *computability* theory

Definition: an algorithm is a TM in the standard representation [a]

---

[a] see A. I. Mal'cev, *Algorithms and recursive functions*, Walter-Noordhoff Publishing, Groningen, The Netherlands, for other defi nitions

Our standard

- The input to a Turing machine is always a string

- If we want an object, other than a string as input, we must fi rst represent that object as a string

- **Note:** strings can easily represent polynomials, graphs, grammars, automata, and any combination of these objects

Encoding objects

- Our notation for encoding an object O into its string representation is $\langle O \rangle$

- If we have several objects O_1, O_2, …, O_k we denote their encoding into a string by $\langle O_1, O_2, \ldots, O_k \rangle$

Note:

1. Encoding itself can be done in many ways. It doesn't matter which encoding we pick because a Turing machine can always translate one encoding into another.

2. The encoding procedure depends upon the domain of application. Hence, we assume that the encoding of an object O is $\langle O \rangle$.

Decoding the input

A Turing machine may be programmed to decode the input representation so that it can be interpreted the way we intend.

Description of a TM

- A Turing machine algorithm is described with an indented segment of text in quotes

- The algorithm is broken into stages, each stage involving many individual steps of computation

- The block structure of the algorithm is indicated by further indentation

- The fi rst line of the algorithm describes the input which is a string w

- If the input is the encoding of an object such as $\langle A \rangle$ the Turing machine fi rst implicitly test

Example TM

- Let A be the language consisting of all strings representing undirected graphs that are connected.

- **Recall:**

  1. A graph G is a tuple $G = (N, E)$ where N is a set of nodes and E is a set of edges, i.e. $E = \{(n_i, n_j) | n_i, n_j \in N\}$;

  2. A graph G is connected if every node can be reached from every other node traveling on edges of the graph.

- **Notation:** $A = \{\langle G \rangle | G \text{ is a connected undirected graph}\}$

A TM deciding A

M = "On input $\langle G \rangle$, the encoding of G

1. Select the first node of G and mark it

2. Repeat the following stage 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 the nodes of G to determine whether they all are marked. If they are *accept*; otherwise *reject*."

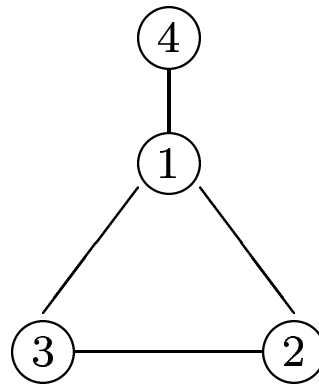# Implementation details

Consider the graph in Figure 1

Figure 1: A connected graph

Graph encoding, $\langle G \rangle$

- The encoding $\langle G \rangle$ of a graph as a string is a list of nodes followed by a list of edges

- Each node is a decimal number, and each edge is a pair of decimal numbers that represent the nodes that edge connects

- **Example encoding:** the graph in Figure 1 is encoded by the string:

$$\langle G \rangle = (1, 2, 3, 4)((1, 2), (2, 3), (3, 1), (1, 4))$$

Checking the encoding

When M receives the input $\langle G \rangle$ it first checks to determine that the input is a proper encoding of some graph:

1. Scan the tape to be sure that there are two lists and that they are in proper form

2. The first list should be a list of distinct decimal numbers; the second list should be a list of pairs of decimal numbers

3. The list of decimal numbers should contain no repetitions

4. Every node on the second list should appear in the first list too.

Note: element distinctness problem can be used to formate the lists and to implement the checks above

More implementation details

- To implement stage 1, M marks the first node with a dot on the leftmost digit.

- To implement stage 2, M scan the list of nodes to find an undotted node n_1 and flags it by marking it differently, example by underlining its first digit.

- Then M scans the list again to find a dotted node n_2 and underlines it too.

Scanning the list of edges

- For each edge, M tests whether the two underlined nodes n_1 and n_2 are the ones appearing in that edge.

- If they are, M dots n_1, removes the underlines, and go one from the beginning of stage 2. If they are not, M checks the next edge on the list.

- If there are no more edges, (n_1, n_2) is not an edge of the graph.

- Then M moves the underline on n_2 to the next dotted node and then call this node the node n_2.

- Repeats the steps above to check, as before, whether the pair (n_1, n_2) is an edge.

Concluding

- If there are no more dotted nodes, n_1 is not attached to any dotted node. Then M sets the underlines so that n_1 is the next undotted node and n_2 is the first dotted node

- Repeats the scanning the list of edges

- If there are no more undotted nodes, M has not been able to find any new nodes to dot, so it moves to stage 4.

Implementing stage 4

- Scan the list of node to determine whether all are dotted.

- If all nodes are dotted, M enters the accept state

- If they are not, M enter the reject state