CS301: Computability and Complexity Theory (CC)

Lecture 3: The Church-turing Thesis

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

October 20, 2023

Table of contents

Section 1

Previously on CS301

TM Formal definition

Definition

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are all finite sets and

1. Q is the set of states
2. Σ is the input alphabet not containing the *blank symbol* $\sqcup$
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state
7. $q_{reject} \in Q$ is the reject state and $q_{accept} \neq q_{reject}$

TM configuration

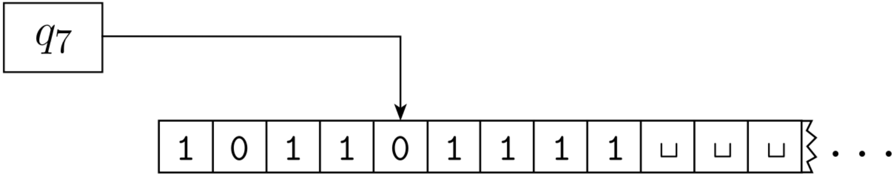


Figure: A TM with configuration $1011q_701111$

Recognizable and decidable

- The collection of strings that M accepts is the language of M or the language recognized by M, denoted $L(M)$
- When we start a TM on an input, three outcomes are possible. The machine may accept, reject, or loop. By loop we mean that the machine simply does not halt
- M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that halt on all inputs; such machines never loop.
- These machines are called *deciders* because they always make a decision to accept or reject. A decider that recognizes some language also is said to decide that language

Recognizable and decidable

Definition

Call a language Turing-recognizable if some Turing machine recognizes it

Definition

Call a language Turing-decidable or simply decidable if some Turing machine decides it

Multitape TMs

- A multitape Turing machine is like an ordinary TM with several tapes
- Each tape has its own head for reading and writing
- Initially the input appears on tape 1, and the others start out blank
- Transition function δ is change to allow reading, writing and moving on some or all of the tapes simultaneously

Multitape TMs

Formalizing we have:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

where k is the number of tapes and the expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1 \ \ldots, b_k, L, R, \ldots, L)$$

means that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k and directs each head to move left or right, or to stay put, as specified.

Multitape TMs

Multitape TMs appear to be more powerful than ordinary TM, but we can show that they are equivalent in power. Recall that two machines are equivalent if they recognize the same language

Theorem

Every multitape Turing machine has an equivalent single-tape Turing machine

Multitape TMs

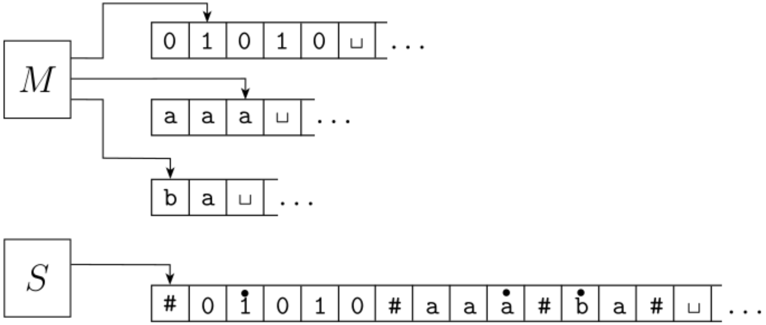


Figure: Representing three tapes with one

Section 2

Context setup

Context setup

Corresponding to Sipser 3.2&3.3

Section 3

Nondeterministic TMs

Nondeterministic TMs

A nondeterministic TM is defined in as follow: At any point in a computation, the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

The computation of a nondeterministic TM is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input.

Nondeterministic TMs

Theorem

Every nondeterministic Turing machine has an equivalent deterministic Turing machine

Proof idea:

- We can simulate any nondeterministic TM N with a deterministic TM D. The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D's simulation will not terminate
- We view N's computation on an input w as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of N. The root of the tree is the start configuration
- The TM D searches this tree for an accepting configuration. Conducting this search carefully is crucial lest D fail to visit the entire tree.

Nondeterministic TMs

Proof idea:

- A bad idea is to have D explore the tree by using depth-first search. The depth-first search strategy goes all the way down one branch before backing up to explore other branches. If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch
- So we design D to explore the tree by using breadth-first search instead. This strategy explores all branches to the same depth before going onto explore any branch to the next depth

Nondeterministic TMs

Proof.

The simulating deterministic TM D has three tapes. By Theorem **??**, this arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way:

- Tape 1 always contains the input string and is never altered
- Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation
- Tape 3 keeps track of D's location in N's nondeterministic computation tree

Nondeterministic TMs

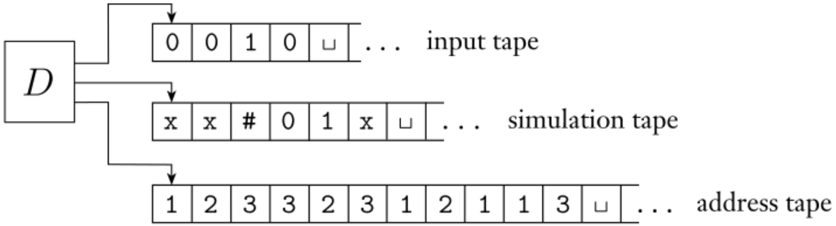


Figure: Deterministic TM D simulating nondeterministic TM N

Nondeterministic TMs

Proof.

Let's look at the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function. To every node in the tree we assign an address that is a string over the alphabet $\Gamma_b = \{1, 2, \ldots, b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case, the address is invalid and doesn't correspond to any node. Tape 3 contains a string over Γ_b. It represents the branch of N's computation from the root to the node addressed by that string unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe D

Nondeterministic TMs

Proof.

D = on input w

1. Initially, tape 1 contains the input w, and tapes 2 and 3 are empty
2. Copy tape 1 to tape 2 and initialize the string on tape 3 to be ϵ
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N, consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input
4. Replace the string on tape 3 with the next string in the string ordering. Simulate the next branch of N's computation by going to stage 2

$\square$

Section 4

The Definition of Algorithm

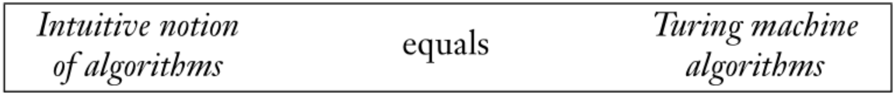
The Definition of Algorithm

- Informally speaking, an algorithm is a collection of simple instructions for carrying out some task
- In everyday life, algorithms sometimes are called procedures or recipes
- Ancient mathematical literature contains descriptions of algorithms for a variety of tasks, such as finding prime numbers and greatest common divisors
- Mathematicians had an intuitive notion of what algorithms were, and relied upon that notion when using and describing them
- The notion of algorithm itself was not defined precisely until the twentieth century
- In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris
- In his lecture, he identified 23 mathematical problems and posed them as a challenge for the coming century. The tenth problem on his list concerned algorithms

The Definition of Algorithm

- Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root
- He did not use the term algorithm but rather **a process according to which it can be determined by a finite number of operations**
- Hilbert explicitly asked that an algorithm be **devised**
- As we now know, no algorithm exists for this task; it is algorithmically unsolvable
- Proving that an algorithm does not exist requires having a clear definition of algorithm
- The definition came in the 1936 papers of Alonzo Church and Alan Turing
- Church used a notational system called the λ-**calculus** to define algorithms. Turing did it with his machines
- These two definitions were shown to be equivalent
- This connection between the informal notion of algorithm and the precise definition has come to be called **the Church–Turing thesis**

The Definition of Algorithm



Intuitive notion of algorithms	equals	*Turing machine algorithms*

Figure: The Church–Turing thesis

The Definition of Algorithm

- The Church–Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem.
- In 1970, Yuri Matijasevic, building on the work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots
- In latter lectures we study the techniques that form the basis for proving that this and other problems are algorithmically unsolvable

Hilbert's tenth problem

Let's phrase Hilbert's tenth problem in our new terminology

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

Hilbert's tenth problem asks in essence whether the set D is decidable. The answer is negative. Instead, we can show that D is Turing-recognizable. First we start with a special case, polynomials having only a single variable

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

A TM M_1 that recognizes D_1:
M1 $=$ On input $< p >$: where p is a polynomial over the variable x.

1. Evaluate p with x set successively to the values $0, 1, -1, 2, -2, 3, -3, \dots$. If at any point the polynomial evaluates to 0, accept.

If p has an integral root, M_1 eventually will find it and accept. If p does not have an integral root, M_1 will run forever

Section 5

Describing TMs

Describing TMs

- We have come to a turning point in the study of the theory of computation. We continue to speak of TM, but our real focus from now on is on algorithms
- The TM merely serves as a precise model for the definition of algorithm
- What is the right level of detail to give when describing such algorithms ?

Describing TMs

There are three possibilities:

1. The first is the **formal description** that spells out in full the TM's states, transition function, . . .
2. Second is a higher level of description, called the **implementation description**, in which we use English prose to describe the way that the TM moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function
3. The third is the **high-level description**, wherein we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head

Describing TMs

- From now on we use **high-level description**
- The input to a TM is always a string
- If we want to provide an object other than a string as input, we must first represent that object as a string
- Our notation for the encoding of an object O into its representation as a string is $< O >$
- If we have several objects $O_1, O_2, \ldots, O_k$ representing string is $< O_1, O_2, \ldots, O_k >$

Describing TMs

- A TM may be programmed to decode the representation so that it can be interpreted in the way we intend
- We describe Turing machine algorithms with an indented segment of text
- We break the algorithm into stages, each usually involving many individual steps of the TM's computation
- The first line of the algorithm describes the input to the machine. If the input description is simply w, the input is taken to be a string
- If the input description is the encoding of an object as in $< A >$, the TM first implicitly tests whether the input properly encodes an object of the desired form and rejects it if it doesn't

Example

A graph is *connected* if every node can be reached from every other node by traveling along the edges of the graph. Let A be the language consisting of all strings representing undirected graphs that are connected.

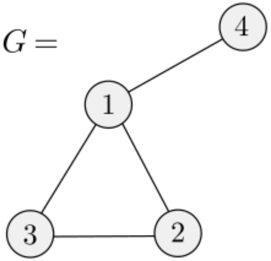
$$A = \{< G > | G \ \textit{is a connected undirected graph}\}$$

M is a high-level description of a TM that decide A
$M =$ On input $< G >$, the encoding of a graph 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*

Example

$G =$



$\langle G \rangle =$

```
(1,2,3,4)((1,2),(2,3),(3,1),(1,4))
```

Figure: A graph G and its encoding $< G >$

Example

- G is encode as a string $< G >$ which is a list of the nodes of G followed by a list of the edges of G
- Each node is a decimal number, and each edge is the pair of decimal numbers that represent the nodes at the two endpoints of the edge
- When M receives input $< G >$ it first checks to determine whether the input is the proper encoding of some graph
- To do so, M scans the tape to be sure that there are two lists and that they are in the proper form
- The first list should be a list of distinct decimal numbers, and the second should be a list of pairs of decimal numbers
- The node list should contain no repetitions; and second, every node appearing on the edge list should also appear on the node list

Example

- For stage 1, M marks the first node with a dot on the leftmost digit
- For stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by underlining it
- Then M scans the list again to find a dotted node n_2 and underlines it, too
- Now M scans 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 goes on from the beginning of stage 2.
- If they aren't, M checks the next edge on the list. If there are no more edges, (n_1, n_2) is not an edge of G
- Repeat the above steps for remaining pairs (n_1, n_2) by selecting another the rest of undotted nodes
- For stage 4, M scans the list of nodes to determine whether all are dotted. If they are, it enters the accept state; otherwise, it enters the reject state
- This completes the description of M