Teoria degli Algoritmi

Corso di Laurea Magistrale in Matematica Applicata

a.a. 2020-21

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

Lecture 2: Turing Machines

Table of Contents

1. Turing Machines
2. Computable Functions
3. Variants of Turing Machines
4. Universal Turing Machine
5. Summary

SAPIENZA
UNIVERSITÀ DI ROMA

Table of Contents

1. Turing Machines
2. Computable Functions
3. Variants of Turing Machines
4. Universal Turing Machine
5. Summary

Our Model of Computation: Turing Machines

- In his famous 1936 paper[1], Alan Turing proposed his own **model of computation**, a.k.a. **Turing Machines** (TMs)

[1] *"On Computable Numbers, with an Application to the Entscheidungsproblem"*

Our Model of Computation: Turing Machines

- In his famous 1936 paper[1], Alan Turing proposed his own **model of computation**, a.k.a. **Turing Machines** (TMs)
- This was an attempt to formally capture all the functions that can be computed by human "computers" following a well-defined set of rules

[1] *"On Computable Numbers, with an Application to the Entscheidungsproblem"*

Our Model of Computation: Turing Machines

- In his famous 1936 paper[1], Alan Turing proposed his own **model of computation**, a.k.a. **Turing Machines** (TMs)
- This was an attempt to formally capture all the functions that can be computed by human "computers" following a well-defined set of rules
- Part of the impetus for the drive to define what is computable came from the mathematician David Hilbert

[1] "*On Computable Numbers, with an Application to the Entscheidungsproblem*"

Our Model of Computation: Turing Machines

- In his famous 1936 paper[1], Alan Turing proposed his own **model of computation**, a.k.a. **Turing Machines** (TMs)
- This was an attempt to formally capture all the functions that can be computed by human "computers" following a well-defined set of rules
- Part of the impetus for the drive to define what is computable came from the mathematician David Hilbert
    - Hilbert wondered if it exists an "effective procedure" (i.e., our informal definition of algorithm) that decides whether any mathematical statement is true or false, in a finite number of steps

[1] *"On Computable Numbers, with an Application to the Entscheidungsproblem"*

Our Model of Computation: Turing Machines

- In his famous 1936 paper[1], Alan Turing proposed his own **model of computation**, a.k.a. **Turing Machines** (TMs)
- This was an attempt to formally capture all the functions that can be computed by human "computers" following a well-defined set of rules
- Part of the impetus for the drive to define what is computable came from the mathematician David Hilbert
  - Hilbert wondered if it exists an "effective procedure" (i.e., our informal definition of algorithm) that decides whether any mathematical statement is true or false, in a finite number of steps
  - As a special case of this decision problem, Hilbert considered the validity problem for first-order logic (a.k.a. *entscheidungsproblem*)

[1] *"On Computable Numbers, with an Application to the Entscheidungsproblem"*

Turing Machines: An Informal Perspective

- To describe his machine, Turing thought of a person as having access to as much "paper" as they need (i.e., simulating infinite memory)

Turing Machines: An Informal Perspective

- To describe his machine, Turing thought of a person as having access to as much "paper" as they need (i.e., simulating infinite memory)
- We can think of this paper as a one-dimensional **tape**

Turing Machines: An Informal Perspective

- To describe his machine, Turing thought of a person as having access to as much "paper" as they need (i.e., simulating infinite memory)
- We can think of this paper as a one-dimensional **tape**
- The tape is divided into "cells", where each cell can hold a single symbol (i.e., some element of a finite alphabet)

Turing Machines: An Informal Perspective

- To describe his machine, Turing thought of a person as having access to as much "paper" as they need (i.e., simulating infinite memory)
- We can think of this paper as a one-dimensional **tape**
- The tape is divided into "cells", where each cell can hold a single symbol (i.e., some element of a finite alphabet)
- At any point in time, the person can read from and write to a single cell of the paper

Turing Machines: An Informal Perspective

- To describe his machine, Turing thought of a person as having access to as much "paper" as they need (i.e., simulating infinite memory)
- We can think of this paper as a one-dimensional **tape**
- The tape is divided into "cells", where each cell can hold a single symbol (i.e., some element of a finite alphabet)
- At any point in time, the person can read from and write to a single cell of the paper
- Based on the content of a cell, the person can update their finite (mental) state, and/or move to the cell immediately to the left or right of the current one

Turing Machines: An Informal Perspective

- To describe his machine, Turing thought of a person as having access to as much "paper" as they need (i.e., simulating infinite memory)
- We can think of this paper as a one-dimensional **tape**
- The tape is divided into "cells", where each cell can hold a single symbol (i.e., some element of a finite alphabet)
- At any point in time, the person can read from and write to a single cell of the paper
- Based on the content of a cell, the person can update their finite (mental) state, and/or move to the cell immediately to the left or right of the current one

Note

The linear nature of memory tape, as opposed to random access memory, is a limitation on computation speed but not power: a TM can find any memory location, i.e., tape cell, by sequentially scanning its tape

Turing Machines: A Formal Definition

Definition (Turing machine)

A Turing machine M is a 6-tuple $(Q, \Sigma, \delta_M, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is the finite set of **states**

Turing Machines: A Formal Definition

Definition (Turing machine)

A Turing machine M is a 6-tuple $(Q, \Sigma, \delta_M, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is the finite set of **states**
- $\Sigma \cup \{\varnothing\}$, where Σ is the finite **alphabet** and $\varnothing$ is the special **blank symbol**

Turing Machines: A Formal Definition

Definition (Turing machine)

A Turing machine M is a 6-tuple $(Q, \Sigma, \delta_M, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is the finite set of **states**
- $\Sigma \cup \{\varnothing\}$, where Σ is the finite **alphabet** and $\varnothing$ is the special **blank symbol**
- $\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\}$ is the **transition function**

Turing Machines: A Formal Definition

Definition (Turing machine)

A Turing machine M is a 6-tuple $(Q, \Sigma, \delta_M, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is the finite set of **states**
- $\Sigma \cup \{\varnothing\}$, where Σ is the finite **alphabet** and $\varnothing$ is the special **blank symbol**
- $\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\}$ is the **transition function**
- $q_0 \in Q$ is the **start state**

Turing Machines: A Formal Definition

Definition (Turing machine)

A Turing machine M is a 6-tuple $(Q, \Sigma, \delta_M, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is the finite set of **states**
- $\Sigma \cup \{\varnothing\}$, where Σ is the finite **alphabet** and $\varnothing$ is the special **blank symbol**
- $\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\}$ is the **transition function**
- $q_0 \in Q$ is the **start state**
- $q_{\text{accept}} \in Q$ is the **accept state**

Turing Machines: A Formal Definition

Definition (Turing machine)

A Turing machine M is a 6-tuple $(Q, \Sigma, \delta_M, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is the finite set of **states**
- $\Sigma \cup \{\varnothing\}$, where Σ is the finite **alphabet** and $\varnothing$ is the special **blank symbol**
- $\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\}$ is the **transition function**
- $q_0 \in Q$ is the **start state**
- $q_{\text{accept}} \in Q$ is the **accept state**
- $q_{\text{reject}} \in Q$ is the **reject state**, s.t. $q_{\text{accept}} \neq q_{\text{reject}}$

Turing Machine: How Does It Work?

- The generic Turing machine M receives its input on the tape, i.e., $\sigma_{\text{in}} \in \Sigma^*$

Turing Machine: How Does It Work?

- The generic Turing machine M receives its input on the tape, i.e., $\sigma_{\text{in}} \in \Sigma^*$
- The rest of the tape is filled with blank symbols ($\varnothing$)

Turing Machine: How Does It Work?

- The generic Turing machine M receives its input on the tape, i.e., $\sigma_{\text{in}} \in \Sigma^*$
- The rest of the tape is filled with blank symbols ($\varnothing$)
- The read/write head starts from the leftmost cell of the tape (e.g., by convention indicated with position $h = 0$)

Turing Machine: How Does It Work?

- The generic Turing machine M receives its input on the tape, i.e., $\sigma_{\text{in}} \in \Sigma^*$
- The rest of the tape is filled with blank symbols ($\varnothing$)
- The read/write head starts from the leftmost cell of the tape (e.g., by convention indicated with position $h = 0$)
- The head keeps scanning the cells on the tape, according to the transition function δ_M

SAPIENZA UNIVERSITÀ DI ROMA

Turing Machine: How Does It Work?

- The generic Turing machine M receives its input on the tape, i.e., $\sigma_{\text{in}} \in \Sigma^*$
- The rest of the tape is filled with blank symbols ($\varnothing$)
- The read/write head starts from the leftmost cell of the tape (e.g., by convention indicated with position $h = 0$)
- The head keeps scanning the cells on the tape, according to the transition function δ_M
- The computation continues until it either enters q_{accept} or q_{reject} state, otherwise M may run forever

SAPIENZA UNIVERSITÀ DI ROMA

Turing Machine: How Does It Work?

- The generic Turing machine M receives its input on the tape, i.e., $\sigma_{\text{in}} \in \Sigma^*$
- The rest of the tape is filled with blank symbols ($\varnothing$)
- The read/write head starts from the leftmost cell of the tape (e.g., by convention indicated with position $h = 0$)
- The head keeps scanning the cells on the tape, according to the transition function δ_M
- The computation continues until it either enters q_{accept} or q_{reject} state, otherwise M may run forever
- If M ever halts, it will leave the output string on the tape, i.e., $\sigma_{\text{out}} \in \Sigma^*$

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:
  - the new state $q' \in Q$ that M should enter

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:
  - the new state $q' \in Q$ that M should enter
  - the new symbol $\sigma' \in \Sigma \cup \{\varnothing\}$ that should be written in the current tape cell

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:
  - the new state $q' \in Q$ that M should enter
  - the new symbol $\sigma' \in \Sigma \cup \{\varnothing\}$ that should be written in the current tape cell
  - the new head position, $h' = h + d$, where $d \in \{-1, 0, 1\}$

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:
  - the new state $q' \in Q$ that M should enter
  - the new symbol $\sigma' \in \Sigma \cup \{\varnothing\}$ that should be written in the current tape cell
  - the new head position, $h' = h + d$, where $d \in \{-1, 0, 1\}$

Note

One should not confuse the transition function δ_M of a Turing machine M with the function f_M that the machine computes:

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:
  - the new state $q' \in Q$ that M should enter
  - the new symbol $\sigma' \in \Sigma \cup \{\varnothing\}$ that should be written in the current tape cell
  - the new head position, $h' = h + d$, where $d \in \{-1, 0, 1\}$

Note

One should not confuse the transition function δ_M of a Turing machine M with the function f_M that the machine computes:

- δ_M is a **finite** function, which takes $|Q| \cdot |\Sigma \cup \{\varnothing\}|$ possible inputs and produces $3 \cdot |Q| \cdot |\Sigma \cup \{\varnothing\}|$ possible outputs

Turing Machine: The Transition Function δ_M

- From any given state $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ and the symbol in the current head position h, δ_M specifies:
  - the new state $q' \in Q$ that M should enter
  - the new symbol $\sigma' \in \Sigma \cup \{\varnothing\}$ that should be written in the current tape cell
  - the new head position, $h' = h + d$, where $d \in \{-1, 0, 1\}$

Note

One should not confuse the transition function δ_M of a Turing machine M with the function f_M that the machine computes:

- δ_M is a **finite** function, which takes $|Q| \cdot |\Sigma \cup \{\varnothing\}|$ possible inputs and produces $3 \cdot |Q| \cdot |\Sigma \cup \{\varnothing\}|$ possible outputs
- The machine can compute an **infinite** function f_M that takes as input a string $\sigma_{\text{in}} \in \Sigma^*$ and produces another string $\sigma_{\text{out}} \in \Sigma^*$ as output, both of arbitrary lengths

Table of Contents

1. Turing Machines
2. Computable Functions
3. Variants of Turing Machines
4. Universal Turing Machine
5. Summary

From Turing Machines to Computable Functions

Definition (Computable Function)

Let $f : \Sigma^* \mapsto \Sigma^*$ be a (total) function and let M be a Turing machine. We say that M computes f if for every $x \in \Sigma^*$, $M(x) = f(x)$.
We say that a function f is computable if there exists a Turing machine M that computes it.

From Turing Machines to Computable Functions

Definition (Computable Function)

Let $f : \Sigma^* \mapsto \Sigma^*$ be a (total) function and let M be a Turing machine. We say that M computes f if for every $x \in \Sigma^*$, $M(x) = f(x)$.
We say that a function f is computable if there exists a Turing machine M that computes it.

Note

Defining a function "computable" if it can be computed by a Turing machine might seem incautious, but this is equivalent to being computable in virtually *any* reasonable model of computation.

The Church-Turing Thesis

- A hypothesis about the nature of computable functions

The Church-Turing Thesis

- A hypothesis about the nature of computable functions
- A function can be calculated by an effective method if and only if it is computable by a Turing machine
  - Or by any equivalent computational models proposed by Gödel (**recursive functions**) and Church (**λ-calculus**)

The Church-Turing Thesis

- A hypothesis about the nature of computable functions
- A function can be calculated by an effective method if and only if it is computable by a Turing machine
  - Or by any equivalent computational models proposed by Gödel (**recursive functions**) and Church (**λ-calculus**)
- The three formally-defined classes of computable functions coincide with the informal notion of an effectively calculable function

The Church-Turing Thesis

- A hypothesis about the nature of computable functions
- A function can be calculated by an effective method if and only if it is computable by a Turing machine
  - Or by any equivalent computational models proposed by Gödel (**recursive functions**) and Church (**λ-calculus**)
- The three formally-defined classes of computable functions coincide with the informal notion of an effectively calculable function
- Since the concept of effective calculability does not have a formal definition, the thesis, although it has near-universal acceptance, cannot be formally proven

(Boolean) Computable Functions

- Often, we are interested in functions $f : \Sigma^* \mapsto \Sigma$, i.e., those having a single bit of output

(Boolean) Computable Functions

- Often, we are interested in functions $f : \Sigma^* \mapsto \Sigma$, i.e., those having a single bit of output
- We can give a special name for the set of such boolean computable functions

(Boolean) Computable Functions

- Often, we are interested in functions $f : \Sigma^* \mapsto \Sigma$, i.e., those having a single bit of output
- We can give a special name for the set of such boolean computable functions

Definition

We define by $\mathcal{R}$ the set of **all** computable functions $f : \Sigma^* \mapsto \Sigma$

Functions vs. Languages

- Many texts use the terminology of "languages" rather than functions to refer to computational tasks

Functions vs. Languages

- Many texts use the terminology of "languages" rather than functions to refer to computational tasks
- A Turing machine M **decides** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$ (0, otherwise)

Functions vs. Languages

- Many texts use the terminology of "languages" rather than functions to refer to computational tasks
- A Turing machine M **decides** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$ (0, otherwise)
- This is equivalent to computing the boolean (total) function $f : \Sigma^* \mapsto \Sigma$ defined as:

$$f(x) = \begin{cases} 1, & \text{if } x \in L \\ 0, & \text{otherwise} \end{cases}$$

Functions vs. Languages

- Many texts use the terminology of "languages" rather than functions to refer to computational tasks
- A Turing machine M **decides** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$ (0, otherwise)
- This is equivalent to computing the boolean (total) function $f : \Sigma^* \mapsto \Sigma$ defined as:

$$f(x) = \begin{cases} 1, & \text{if } x \in L \\ 0, & \text{otherwise} \end{cases}$$

Definition (Turing-decidable Language)

A language L is **Turing-decidable** (or simply **decidable**) if there is a Turing machine M that decides it

A Note on the Terminology

- For historical reasons, some texts also refer to computable boolean functions/decidable languages as **recursive languages**

A Note on the Terminology

- For historical reasons, some texts also refer to computable boolean functions/decidable languages as **recursive languages**
- This is also the reason why the letter $\mathcal{R}$ is often used

A Note on the Terminology

- For historical reasons, some texts also refer to computable boolean functions/decidable languages as **recursive languages**
- This is also the reason why the letter $\mathcal{R}$ is often used
- We stick to the term *functions* rather than *lanuguages*, although the following always holds:

$$f : \Sigma^* \mapsto \Sigma$$

$$L = \{x \in \Sigma^* \mid f(x) = 1\}$$

Infinite Loops and Partial Functions

- Given a Turing machine M, we cannot determine a priori the length of its output

Infinite Loops and Partial Functions

- Given a Turing machine M, we cannot determine a priori the length of its output
- In fact, we don't even know if an output will be produced at all!

Infinite Loops and Partial Functions

- Given a Turing machine M, we cannot determine a priori the length of its output
- In fact, we don't even know if an output will be produced at all!
  - For example, it is easy to design a Turing machine whose transition function never leads to a halting state (i.e., either q_{accept} or q_{reject})

Infinite Loops and Partial Functions

- Given a Turing machine M, we cannot determine a priori the length of its output
- In fact, we don't even know if an output will be produced at all!
  - For example, it is easy to design a Turing machine whose transition function never leads to a halting state (i.e., either q_{accept} or q_{reject})
- If a machine M fails to stop and produce an output on some input $x \in \Sigma^*$, then it cannot compute any total function f

Infinite Loops and Partial Functions

- Given a Turing machine M, we cannot determine a priori the length of its output
- In fact, we don't even know if an output will be produced at all!
  - For example, it is easy to design a Turing machine whose transition function never leads to a halting state (i.e., either q_{accept} or q_{reject})
- If a machine M fails to stop and produce an output on some input $x \in \Sigma^*$, then it cannot compute any total function f
- However, M can still compute a **partial function**

Infinite Loops and Partial Functions

- Given a Turing machine M, we cannot determine a priori the length of its output
- In fact, we don't even know if an output will be produced at all!
  - For example, it is easy to design a Turing machine whose transition function never leads to a halting state (i.e., either q_{accept} or q_{reject})
- If a machine M fails to stop and produce an output on some input $x \in \Sigma^*$, then it cannot compute any total function f
- However, M can still compute a **partial function**

Definition

A partial function $f : A \mapsto B$ is a function that is only defined on a subset A' of A (i.e., $A' \subset A$). We can also think of such a function as mapping from A to $B \cup \{\perp\}$, where $\perp$ is a special "failure" symbol such that $f(a) = \perp$ indicates f is not defined on input a

Turing Machines Computing Partial Functions

Example

Consider the function $div : \mathbb{Z}^{0+} \times \mathbb{Z}^{0+} \mapsto \mathbb{Z}^{0+}$, defined as follows:

$$div(a, b) = \begin{cases} \left\lceil \frac{a}{b} \right\rceil, & \text{if } b > 0 \\ \perp, & \text{otherwise} \end{cases}$$

Turing Machines Computing Partial Functions

Example

Consider the function $div : \mathbb{Z}^{0+} \times \mathbb{Z}^{0+} \mapsto \mathbb{Z}^{0+}$, defined as follows:

$$div(a, b) = \begin{cases} \lceil \frac{a}{b} \rceil, & \text{if } b > 0 \\ \perp, & \text{otherwise} \end{cases}$$

- We can design a Turing machine M that computes div on inputs a, b by outputting the first $c \in \{0, 1, 2, \ldots\}$ such that $cb \geq a$

Turing Machines Computing Partial Functions

Example

Consider the function $div : \mathbb{Z}^{0+} \times \mathbb{Z}^{0+} \mapsto \mathbb{Z}^{0+}$, defined as follows:

$$div(a, b) = \begin{cases} \lceil \frac{a}{b} \rceil, & \text{if } b > 0 \\ \perp, & \text{otherwise} \end{cases}$$

- We can design a Turing machine M that computes div on inputs a, b by outputting the first $c \in \{0, 1, 2, \ldots\}$ such that $cb \geq a$
  - If $a > 0$ and $b = 0$, M never halts but this is ok, since div is undefined on such inputs

Turing Machines Computing Partial Functions

Example

Consider the function $div : \mathbb{Z}^{0+} \times \mathbb{Z}^{0+} \mapsto \mathbb{Z}^{0+}$, defined as follows:

$$div(a, b) = \begin{cases} \left\lceil \frac{a}{b} \right\rceil, & \text{if } b > 0 \\ \perp, & \text{otherwise} \end{cases}$$

- We can design a Turing machine M that computes div on inputs a, b by outputting the first $c \in \{0, 1, 2, \ldots\}$ such that $cb \geq a$
  - If $a > 0$ and $b = 0$, M never halts but this is ok, since div is undefined on such inputs
  - If $a = b = 0$, M will output 0, which is also ok, since we do not care about what the machine outputs on inputs on which div is undefined

SAPIENZA UNIVERSITÀ DI ROMA

Computable Functions (Redefined)

Definition

Let f be a **total** or **partial** function, such that $f : \Sigma^* \mapsto \Sigma^*$ and let M be a Turing machine.
We say that M **computes** f if for every $x \in \Sigma^*$ on which f is defined, $M(x) = f(x)$.
We say that a (partial or total) function f is **computable** if there is a Turing machine that computes it.

A Clarification on the Role of $\perp$

- We used $\perp$ as our special "failure symbol"; if a Turing machine M fails to halt on some input $x \in \Sigma^*$ then we denote this by $M(x) = \perp$

A Clarification on the Role of $\perp$

- We used $\perp$ as our special "failure symbol"; if a Turing machine M fails to halt on some input $x \in \Sigma^*$ then we denote this by $M(x) = \perp$
- This **does not** mean that M outputs some encoding of the symbol $\perp$ but rather that M enters into an infinite loop when given x as input

A Clarification on the Role of $\perp$

- We used $\perp$ as our special "failure symbol"; if a Turing machine M fails to halt on some input $x \in \Sigma^*$ then we denote this by $M(x) = \perp$
- This **does not** mean that M outputs some encoding of the symbol $\perp$ but rather that M enters into an infinite loop when given x as input
- As such, one might be tempted to think that M halts on x if and only if f is defined on x

A Clarification on the Role of $\perp$

- We used $\perp$ as our special "failure symbol"; if a Turing machine M fails to halt on some input $x \in \Sigma^*$ then we denote this by $M(x) = \perp$
- This **does not** mean that M outputs some encoding of the symbol $\perp$ but rather that M enters into an infinite loop when given x as input
- As such, one might be tempted to think that M halts on x if and only if f is defined on x
- However, for a Turing machine M to compute a partial function f it is **not** necessary to enter an infinite loop on inputs x outside the domain of f

SAPIENZA UNIVERSITÀ DI ROMA

A Clarification on the Role of $\perp$

- We used $\perp$ as our special "failure symbol"; if a Turing machine M fails to halt on some input $x \in \Sigma^*$ then we denote this by $M(x) = \perp$
- This **does not** mean that M outputs some encoding of the symbol $\perp$ but rather that M enters into an infinite loop when given x as input
- As such, one might be tempted to think that M halts on x if and only if f is defined on x
- However, for a Turing machine M to compute a partial function f it is **not** necessary to enter an infinite loop on inputs x outside the domain of f
- All that is needed is for M to output $f(x)$ on $x \in domain(f)$: on any other input it is OK for M to output an arbitrary value or not to halt at all

Functions vs. Languages

- A Turing machine M **recognizes** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$

Functions vs. Languages

- A Turing machine M **recognizes** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$
- If $x \notin L$, M may either halt with non-sense output or loop forever

Functions vs. Languages

- A Turing machine M **recognizes** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$
- If $x \notin L$, M may either halt with non-sense output or loop forever
- This is equivalent to computing the partial function $f : \Sigma^* \mapsto \Sigma$ defined as:

$$f(x) = \begin{cases} 1, & \text{if } x \in L \\ \perp, & \text{otherwise} \end{cases}$$

Functions vs. Languages

- A Turing machine M **recognizes** a language L if for **every** input $x \in \Sigma^*$, $M(x)$ outputs 1 if and only if $x \in L$
- If $x \notin L$, M may either halt with non-sense output or loop forever
- This is equivalent to computing the partial function $f : \Sigma^* \mapsto \Sigma$ defined as:

$$f(x) = \begin{cases} 1, & \text{if } x \in L \\ \perp, & \text{otherwise} \end{cases}$$

Definition (Turing-recognizable Language)

A language L is **Turing-recognizable** (or simply **recognizable** or **semi-decidable**) if there is a Turing machine M that recognizes it

A Note on the Terminology

- For historical reasons, some texts also refer to recognizable languages as **recursively enumerable languages**

A Note on the Terminology

- For historical reasons, some texts also refer to recognizable languages as **recursively enumerable languages**
- This is also the reason why the letter $\mathcal{RE}$ is often used

A Note on the Terminology

- For historical reasons, some texts also refer to recognizable languages as **recursively enumerable languages**
- This is also the reason why the letter $\mathcal{RE}$ is often used
- We stick to the term *functions* rather than *lanuguages*, although the following always holds:

$$f : \Sigma^* \mapsto \Sigma$$

$$L = \{x \in \Sigma^* \mid f(x) = 1\}$$

Table of Contents

1. Turing Machines
2. Computable Functions
3. Variants of Turing Machines
4. Universal Turing Machine
5. Summary

Variants of Turing Machines

- Alternative definitions of Turing machines abound, e.g., multiple tapes or non-deterministic Turing machines

Variants of Turing Machines

- Alternative definitions of Turing machines abound, e.g., multiple tapes or non-deterministic Turing machines
- Interestingly enough, the original computational model and its variants have all the same power

Variants of Turing Machines

- Alternative definitions of Turing machines abound, e.g., multiple tapes or non-deterministic Turing machines
- Interestingly enough, the original computational model and its variants have all the same power
- They all compute the same functions/recognize the same set of languages

Multi-tape Turing Machines

- Like an ordinary Turing machine, yet with several tapes

Multi-tape Turing Machines

- Like an ordinary Turing machine, yet with several tapes
- Each tape has its own reading/writing head

Multi-tape Turing Machines

- Like an ordinary Turing machine, yet with several tapes
- Each tape has its own reading/writing head
- Initially, the input is located on the first tape (i.e., tape 1), whilst the others are filled with blank symbols

Multi-tape Turing Machines

- Like an ordinary Turing machine, yet with several tapes
- Each tape has its own reading/writing head
- Initially, the input is located on the first tape (i.e., tape 1), whilst the others are filled with blank symbols
- The transition function δ_M is changed to allow for reading, writing, and moving the heads on some or all of the tapes, simultaneously

Multi-tape Turing Machines

- Like an ordinary Turing machine, yet with several tapes
- Each tape has its own reading/writing head
- Initially, the input is located on the first tape (i.e., tape 1), whilst the others are filled with blank symbols
- The transition function δ_M is changed to allow for reading, writing, and moving the heads on some or all of the tapes, simultaneously
- Formally, the transition function of a k-tape Turing machine is defined as follows:

$$\delta_M : Q \times \Sigma^k \cup \{\varnothing\}^k \mapsto Q \times \Sigma^k \cup \{\varnothing\}^k \times \{-1, 0, +1\}^k$$

Multi-tape Turing Machines: Example

Consider a k-tape Turing Machine, then the expression

$$\delta_M(q_i, \sigma_1, \sigma_2, \ldots, \sigma_k) = (q_j, \sigma'_1, \sigma'_2, \ldots, \sigma'_k, +1, 0, \ldots, -1)$$

means that, if the machine is in state q_i and heads 1 through k are reading symbols σ_1 through σ_k, then it goes to state q_j, writes symbols σ'_1 through σ'_k and moves each head to the left (-1) or to the right (+1) of the current position, or leaves it where it is (0)

Equivalence Between Single- and Multi-Tape TMs

- Intuitively, multi-tape Turing machines seem more powerful than ordinary, single-tape Turing machines

Equivalence Between Single- and Multi-Tape TMs

- Intuitively, multi-tape Turing machines seem more powerful than ordinary, single-tape Turing machines
- In fact, it can be proven that those two models of computations are indeed equivalent (i.e., they both recognize the same languages)

Equivalence Between Single- and Multi-Tape TMs

- Intuitively, multi-tape Turing machines seem more powerful than ordinary, single-tape Turing machines
- In fact, it can be proven that those two models of computations are indeed equivalent (i.e., they both recognize the same languages)
- To sketch the idea of the proof, consider two Turing machines: S, M
  - The former is a single-tape machine, whilst the latter is multi-tape
  - The key idea is to simulate M using S
  - We can lay down the content of the k tapes of M on the single tape of S, using a special symbol as delimiter (e.g., $\#$)
  - Add another extra symbol (e.g., $\bullet$) on top of the current symbol to mimic the head position on each tape

Non-deterministic Turing Machines (NTMs)

- At any time during the computation a non-deterministic TM proceeds according to several possibilities

Non-deterministic Turing Machines (NTMs)

- At any time during the computation a non-deterministic TM proceeds according to several possibilities
- The transition function for a NTM δ_M is defined as follows:

$$\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto \mathcal{P}(Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\})$$

where $\mathcal{P}(A)$ stands for the **power set** of A, i.e., the set of all subsets of A

Non-deterministic Turing Machines (NTMs)

- At any time during the computation a non-deterministic TM proceeds according to several possibilities
- The transition function for a NTM δ_M is defined as follows:

$$\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto \mathcal{P}(Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\})$$

  where $\mathcal{P}(A)$ stands for the **power set** of A, i.e., the set of all subsets of A
- The computation of an NTM is a **tree**, whose branches correspond to different computational paths for the machine

Non-deterministic Turing Machines (NTMs)

- At any time during the computation a non-deterministic TM proceeds according to several possibilities
- The transition function for a NTM δ_M is defined as follows:

$$\delta_M : Q \times \Sigma \cup \{\varnothing\} \mapsto \mathcal{P}(Q \times \Sigma \cup \{\varnothing\} \times \{-1, 0, +1\})$$

  where $\mathcal{P}(A)$ stands for the **power set** of A, i.e., the set of all subsets of A
- The computation of an NTM is a **tree**, whose branches correspond to different computational paths for the machine
- If some branch leads to the accept state (q_{accept}), the machine accepts its input

Equivalence Between Deterministic and Non-Deterministic TMs

- Again, intuitively NTMs seem more powerful than ordinary, deterministic TMs

Equivalence Between Deterministic and Non-Deterministic TMs

- Again, intuitively NTMs seem more powerful than ordinary, deterministic TMs
- In fact, it can be proven that those two models of computations are indeed equivalent (i.e., they both recognize the same languages)

Equivalence Between Deterministic and Non-Deterministic TMs

- Again, intuitively NTMs seem more powerful than ordinary, deterministic TMs
- In fact, it can be proven that those two models of computations are indeed equivalent (i.e., they both recognize the same languages)
- To sketch the idea of the proof, consider two Turing machines: D, N
  - The former is a deterministic machine, whilst the latter is non-deterministic
  - The key idea is to simulate N using D by letting D try **all** the possible branches of N's non-deterministic computation
  - If D ever reaches the accept state on one of these branches, D accepts; otherwise D's simulation may run forever

Non-Deterministic Computation as a Tree

- We can view N's computation on an input string x as a tree

Non-Deterministic Computation as a Tree

- We can view N's computation on an input string x as a tree
- Each node of such a tree is a configuration of N, with the **root node** being the starting configuration

Non-Deterministic Computation as a Tree

- We can view N's computation on an input string x as a tree
- Each node of such a tree is a configuration of N, with the **root node** being the starting configuration
- The machine D searches this tree for an accepting configuration (i.e., a configuration whose state is q_{accept})

Non-Deterministic Computation as a Tree

- We can view N's computation on an input string x as a tree
- Each node of such a tree is a configuration of N, with the **root node** being the starting configuration
- The machine D searches this tree for an accepting configuration (i.e., a configuration whose state is q_{accept})
- **breadth first search** explores all branches at the same depth of the tree before moving to the next level

Non-Deterministic Computation as a Tree

- We can view N's computation on an input string x as a tree
- Each node of such a tree is a configuration of N, with the **root node** being the starting configuration
- The machine D searches this tree for an accepting configuration (i.e., a configuration whose state is q_{accept})
- **breadth first search** explores all branches at the same depth of the tree before moving to the next level
- This guarantees that D will visit every node in the tree until it encounters an accepting configuration

Table of Contents

1. Turing Machines
2. Computable Functions
3. Variants of Turing Machines
4. Universal Turing Machine
5. Summary

Universal Turing Machine

- So far, we have roughly assumed that a Turing Machine M takes as input some x encoded as a binary string and computes a function $f(x)$

Universal Turing Machine

- So far, we have roughly assumed that a Turing Machine M takes as input some x encoded as a binary string and computes a function $f(x)$
- We have already seen that we can use the same binary string encoding to represent virtually **any** object

Universal Turing Machine

- So far, we have roughly assumed that a Turing Machine M takes as input some x encoded as a binary string and computes a function $f(x)$
- We have already seen that we can use the same binary string encoding to represent virtually **any** object
- As a special case, we can therefore encode **any** Turing machine M together with **any** of its input x

Universal Turing Machine

Definition (Universal Turing Machine)

There exists a Turing machine U, such that on every string M which encodes a Turing machine, and $x \in \Sigma^*$:

$$U(M, x) = M(x)$$

If the machine M halts on x and outputs some $y \in \Sigma^*$ (i.e., $M(x) = y$), then:

$$U(M, x) = M(x) = y$$

If M does **not** halt on x (i.e., $M(x) = \perp$) then:

$$U(M, x) = M(x) = \perp$$

Universal Turing Machine: Intuition

- Intuitively, the existence of U implies the existence of a "universal" algorithm that can evaluate arbitrary algorithms (M) on arbitrary inputs (x)

Universal Turing Machine: Intuition

- Intuitively, the existence of U implies the existence of a “universal” algorithm that can evaluate arbitrary algorithms (M) on arbitrary inputs (x)
- The desired program U is an **interpreter** for Turing machines

Universal Turing Machine: Intuition

- Intuitively, the existence of U implies the existence of a "universal" algorithm that can evaluate arbitrary algorithms (M) on arbitrary inputs (x)
- The desired program U is an **interpreter** for Turing machines
- U gets a representation of the machine M (e.g., source code), and some input x, and simulates the execution of M on x

The Existence of a Universal Turing Machine

- How would you code U in your favorite programming language?

The Existence of a Universal Turing Machine

- How would you code U in your favorite programming language?
- First, you need to decide on some representation scheme for M.
  - For example, you can use an array or a dictionary to encode M's transition function

The Existence of a Universal Turing Machine

- How would you code U in your favorite programming language?
- First, you need to decide on some representation scheme for M.
  - For example, you can use an array or a dictionary to encode M's transition function
- Then you would use some data structure, such as a list, to store the contents of M's tape

The Existence of a Universal Turing Machine

- How would you code U in your favorite programming language?
- First, you need to decide on some representation scheme for M.
  - For example, you can use an array or a dictionary to encode M's transition function
- Then you would use some data structure, such as a list, to store the contents of M's tape
- Now you can simulate M step by step, updating the data structure as you move along

The Existence of a Universal Turing Machine

- How would you code U in your favorite programming language?
- First, you need to decide on some representation scheme for M.
  - For example, you can use an array or a dictionary to encode M's transition function
- Then you would use some data structure, such as a list, to store the contents of M's tape
- Now you can simulate M step by step, updating the data structure as you move along
- The **interpreter** will continue the simulation until the machine eventually halts

The Existence of a Universal Turing Machine

- How would you code U in your favorite programming language?
- First, you need to decide on some representation scheme for M.
  - For example, you can use an array or a dictionary to encode M's transition function
- Then you would use some data structure, such as a list, to store the contents of M's tape
- Now you can simulate M step by step, updating the data structure as you move along
- The **interpreter** will continue the simulation until the machine eventually halts
- Translating the interpreter above into the corresponding Turing machine is "easy"

Universal Turing Machine: Implications

- There is more than one Turing machine U that works as indicated above

Universal Turing Machine: Implications

- There is more than one Turing machine U that works as indicated above
- The existence of even a *single* such machine is already fundamental to computer science

Universal Turing Machine: Implications

- There is more than one Turing machine U that works as indicated above
- The existence of even a *single* such machine is already fundamental to computer science
- The idea of a "universal program" is of course not limited to theory

Universal Turing Machine: Implications

- There is more than one Turing machine U that works as indicated above
- The existence of even a *single* such machine is already fundamental to computer science
- The idea of a "universal program" is of course not limited to theory
- The most famous practical example is represented by **compilers** (for programming languages), which are often used to compile themselves!

Table of Contents

1. Turing Machines
2. Computable Functions
3. Variants of Turing Machines
4. Universal Turing Machine
5. Summary

Summary

- We have discussed Turing machines (TMs) as the standard **model of computation**

Summary

- We have discussed Turing machines (TMs) as the standard **model of computation**
- TMs and every other computational model independently proposed have all the same power (**Church-Turing thesis**)

Summary

- We have discussed Turing machines (TMs) as the standard **model of computation**
- TMs and every other computational model independently proposed have all the same power (**Church-Turing thesis**)
- Computable functions (total/partial) are those which can be computed by a TM

Summary

- We have discussed Turing machines (TMs) as the standard **model of computation**
- TMs and every other computational model independently proposed have all the same power (**Church-Turing thesis**)
- Computable functions (total/partial) are those which can be computed by a TM
- There exists few variants of standard TM like multi-tape or non-deterministic TMs yet they all have the same power

Summary

- We have discussed Turing machines (TMs) as the standard **model of computation**
- TMs and every other computational model independently proposed have all the same power (**Church-Turing thesis**)
- Computable functions (total/partial) are those which can be computed by a TM
- There exists few variants of standard TM like multi-tape or non-deterministic TMs yet they all have the same power
- The existence of a special Universal Turing Machine (UTM) allows us to design an algorithm that can run any other algorithm

SAPIENZA UNIVERSITÀ DI ROMA