Languages and Algorithms for Artificial Intelligence (Third Module)

The Computational Model

Ugo Dal Lago

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Inria informatiques mathématiques

University of Bologna, Academic Year 2019/2020

The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!

The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!
- But how about **negative** results?
  - Which machine should we choose?
  - If we choose one specific, concrete machine without relating to the other ones, then our negative results would be *vacuous*.

The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!
- But how about **negative** results?
  - Which machine should we choose?
  - If we choose one specific, concrete machine without relating to the other ones, then our negative results would be *vacuous*.
- The only way out is to define a **model of computation** in the form of an abstract machine, keeping in mind that:
  - It should be as simple as possible, to *facilitate proofs*.
  - It must be able to simulate with reasonable overhead *all physically realistic* machines.

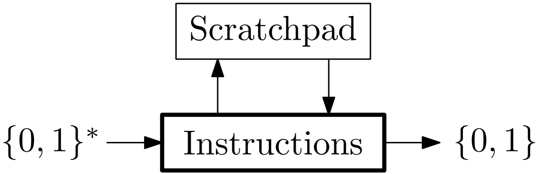
The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!
- But how about **negative** results?
  - Which machine should we choose?
  - If we choose one specific, concrete machine without relating to the other ones, then our negative results would be *vacuous*.
- The only way out is to define a **model of computation** in the form of an abstract machine, keeping in mind that:
  - It should be as simple as possible, to *facilitate proofs*.
  - It must be able to simulate with reasonable overhead *all physically realistic* machines.
- There is a universally accepted model of computation, that we will take as our reference model, namely the **Turing Machine**.
  - This part of the module is specifically about it.

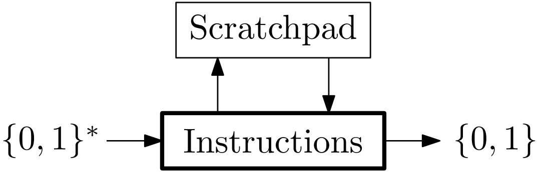
Part I

The Model, Informally

How to Compute $f : \{0,1\}^* \to \{0,1\}$



How to Compute $f : \{0,1\}^* \to \{0,1\}$



- The set of instructions to be followed is fixed (and should work for every input x), and finite.
- The same instruction can be used potentially many times.
- Every instruction proceeds by:
  - Reading a bit of the input;
  - Reading a symbol from the scratchpad;

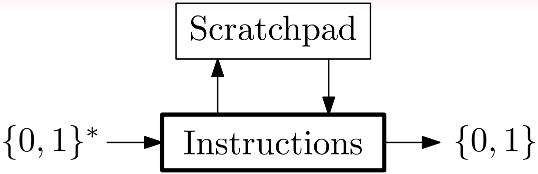
  and based on that decide what to do next, namely:
  - either write symbol to the scratchpad, and proceed to another instruction;
  - or declare the computation finished, by stopping it and outputting either 0 or 1.

How to Compute $f : \{0,1\}^* \to \{0,1\}$

$$\begin{array}{c} \boxed{\text{Scratchpad}} \\ \uparrow \quad \downarrow \\ \{0,1\}^* \longrightarrow \boxed{\text{Instructions}} \longrightarrow \{0,1\} \end{array}$$

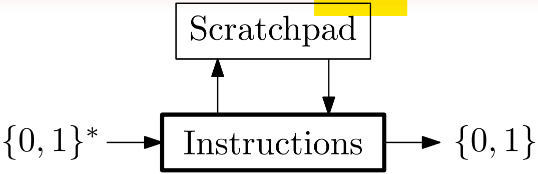
- The **running time** of this machine/process/algorithm on x is simply the number of these basic instructions which are executed on a certain input x.
- We say that the machine **runs in time** $T(n)$ if it performs *at most* $T(n)$ instructions on input strings *of length* n.

How to Compute $f : \{0,1\}^* \to \{0,1\}$



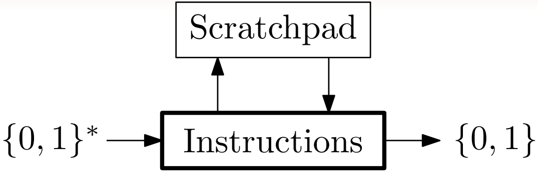
- The **running time** of this machine/process/algorithm on x is simply the number of these basic instructions which are executed on a certain input x.
- We say that the machine **runs in time** $T(n)$ if it performs *at most* $T(n)$ instructions on input strings *of length* n.
- The model is **robust** to many tweaks in the definition, (e.g. changing the alphabet, allowing multiple scratchpads rather than one): the simplest model can simulate the more complicated with *a polynomial overhead* in time.

How to Compute $f : \{0,1\}^* \to \{0,1\}$



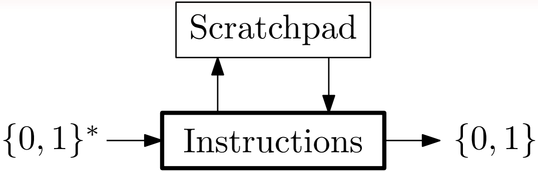
- The **running time** of this machine/process/algorithm on x is simply the number of these basic instructions which are executed on a certain input x.
- We say that the machine **runs in time** $T(n)$ if it performs *at most* $T(n)$ instructions on input strings *of length* n.
- The model is **robust** to many tweaks in the definition, (e.g. changing the alphabet, allowing multiple scratchpads rather than one): the simplest model can simulate the more complicated with *a polynomial overhead* in time.
- Since there are finitely many instructions, machine descriptions can be encoded as **encoded as binary strings** themselves.

How to Compute $f : \{0,1\}^* \to \{0,1\}$



- Given a string α, we indicate as $\mathcal{M}_\alpha$ the Turing Machine α encodes.

How to Compute $f : \{0,1\}^* \to \{0,1\}$

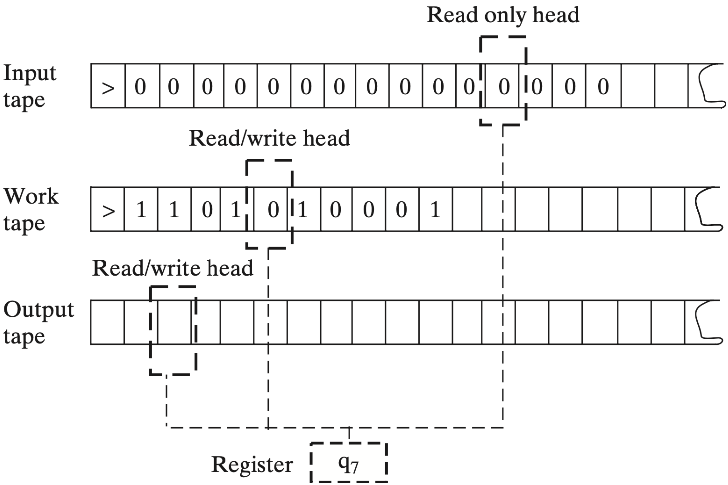


- Given a string α, we indicate as $\mathcal{M}_\alpha$ the Turing Machine α encodes.
- There is a so-called **Universal** Turing Machine $\mathcal{U}$, which simulates any other Turing Machine given its string representation: from a pair of strings (x, α), the machine $\mathcal{U}$ simulates the behaviour of $\mathcal{M}_\alpha$ on x.
  - The simulation is very efficient: if the running time of $\mathcal{M}_\alpha$ were $T(|x|)$, then $\mathcal{U}$ would take time $O(T(|x|) \log T(|x|))$.
- There are functions $f : \{0,1\}^* \to \{0,1\}$ which are **intrinsically uncomputable** by Turing Machines, and this can be proved formally.
  - This has intimate connections to Gödel's famous **incompleteness theorem**.

Part II

The Model, Formally

A More Detailed View



The Scratchpad(s)

- It consists of k **tapes**, where a tape is an infinite one-directional line of cells, each of which can hold a simbol from a finite alphabel Γ, the *alphabet* of the machine.
- Each tape is equipped with **tape head**, which can read or write symbols *from* or *to* the tape. Each head can move left or right.
- The first tape is designated as **input tape**, and is read-only.
- The last tape is the **output tape**, and contains the result of the computation.
  - This slightly deviates from what we have said in our informal account, and is needed to compute arbitrary functions on $\{0, 1\}^*$.

The Instructions

- The machine has a finite set of *states*, called Q, which determine the action to be taken at the next step.
- At *each step*, the machine:
  1. **Read** the symbols under the k tape heads.
  2. For the $k-1$ read-write tapes, **replace** the symbol with a new one, or leave it unchanged.
  3. **Change** its state to a new one.
  4. **Move** each of the k tape heads to the left or to the right (or stay in place).

The Instructions

- The machine has a finite set of *states*, called Q, which determine the action to be taken at the next step.
- At *each step*, the machine:
  1. **Read** the symbols under the k tape heads.
  2. For the $k-1$ read-write tapes, **replace** the symbol with a new one, or leave it unchanged.
  3. **Change** its state to a new one.
  4. **Move** each of the k tape heads to the left or to the right (or stay in place).
- These instructions are of course very basic, and far from being close to the kind of instructions programming languages offer.
  - The point here is to have a *simple*, but *expressive* model.

The Formal Definition

A **Turing Machine** (TM for short) working on k tapes is described as a triple (Γ, Q, δ) containing

- A finite set Γ of **tape symbols**, which we assume contains the *blank symbol* $\square$, the *start symbol* $\triangleright$, and the binary digits 0 and 1.

The Formal Definition

A **Turing Machine** (TM for short) working on k tapes is described as a triple (Γ, Q, δ) containing

- A finite set Γ of **tape symbols**, which we assume contains the *blank symbol* $\square$, the *start symbol* $\triangleright$, and the binary digits 0 and 1.
- A finite set Q of **states** which includes a designated *initial state* $q_{\mathtt{init}}$ and a designated final state $q_{\mathtt{halt}}$.

The Formal Definition

A **Turing Machine** (TM for short) working on k tapes is described as a triple (Γ, Q, δ) containing

- A finite set Γ of **tape symbols**, which we assume contains the *blank symbol* $\Box$, the *start symbol* $\triangleright$, and the binary digits 0 and 1.
- A finite set Q of **states** which includes a designated *initial state* $q_{\mathtt{init}}$ and a designated final state $q_{\mathtt{halt}}$.
- A **transition function**

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$$

  describing the instructions regulating the functioning of the machine at each step.
  - When the first parameter is $q_{\mathtt{halt}}$, then δ cannot touch the tapes nor the heads:

$$\delta(q_{\mathtt{halt}}, (\sigma_1, \ldots, \sigma_k)) = (q_{\mathtt{halt}}, (\sigma_2 \ldots, \sigma_k), (\mathtt{S}, \ldots, \mathtt{S}))$$

    and the machine is *stuck*.

Machine Configurations and Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on k tapes:

- A **configuration** consists of
  - The current state q.
  - The contents of the k tapes.
  - The positions of the k tape heads.

  One such configuration will be indicated as C.

Machine Configurations and Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on k tapes:

- A **configuration** consists of
  - The current state q.
  - The contents of the k tapes.
  - The positions of the k tape heads.

  One such configuration will be indicated as C.
- The **initial configuration** for the input $x \in \{0, 1\}^*$ is the configuration $\mathcal{I}_x$ in which:
  - The current state is $q_{\mathtt{init}}$.
  - The first tape contains $\triangleright x$, followed by blank symbols, while the other tapes contain $\triangleright$, followed by blank symbols.
  - The tape heads are positioned on the first symbol of the k tapes.

Machine Configurations and Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on k tapes:

- A **configuration** consists of
  - The current state q.
  - The contents of the k tapes.
  - The positions of the k tape heads.

  One such configuration will be indicated as C.
- The **initial configuration** for the input $x \in \{0,1\}^*$ is the configuration $\mathcal{I}_x$ in which:
  - The current state is $q_{\mathtt{init}}$.
  - The first tape contains $\triangleright x$, followed by blank symbols, while the other tapes contain $\triangleright$, followed by blank symbols.
  - The tape heads are positioned on the first symbol of the k tapes.
- A **final configuration** for the output $y \in \{0,1\}^*$ is any configuration whose state is $q_{\mathtt{halt}}$ and in which the content of the output tape is $\triangleright y$, followed by blank symbols.

Machine Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on k tapes:

- Given any configuration C, the transition function δ determines in a natural way the next configuration D, and we write $C \xrightarrow{\delta} D$ if this is the case.
- We say that $\mathcal{M}$ **returns** $y \in \{0,1\}^*$ **on input** $x \in \{0,1\}^*$ **in** t **steps** if

$$\mathcal{I}_x \xrightarrow{\delta} C_1 \xrightarrow{\delta} C_2 \xrightarrow{\delta} \ldots \xrightarrow{\delta} C_t$$

  where C_t is a final configuration for y. We write $\mathcal{M}(x)$ for y if this holds.
- Finally, we say that $\mathcal{M}$ **computes** a function $f : \{0,1\}^* \to \{0,1\}^*$ iff $\mathcal{M}(x) = f(x)$ for every $x \in \{0,1\}^*$. In this case, f is said to be **computable**.
  - Beware: for the moment, we do not put any constraint *on the number of steps* $\mathcal{M}$ needs to compute $f(x)$ from x.

The Expressive Power of TMs

- Please take a loot at

  `http://turingmachinesimulator.com`

- Explicitly constructing TMs is **very tedious**.
  - One needs to give the set of states, the set of instructions, etc.
  - One also needs to *prove* that the construction is correct.

The Expressive Power of TMs

- Please take a loot at

  `http://turingmachinesimulator.com`

- Explicitly constructing TMs is **very tedious**.
  - One needs to give the set of states, the set of instructions, etc.
  - One also needs to *prove* that the construction is correct.
- Usually, functions on binary strings are shown to be computable by informally describing *algorithms* or *programs* computing the function.
  - This is fine, *provided* program or algorithm instructions can be **simulated** by TMs.

The Expressive Power of TMs

- Please take a loot at

  `http://turingmachinesimulator.com`

- Explicitly constructing TMs is **very tedious**.
  - One needs to give the set of states, the set of instructions, etc.
  - One also needs to *prove* that the construction is correct.
- Usually, functions on binary strings are shown to be computable by informally describing *algorithms* or *programs* computing the function.
  - This is fine, *provided* program or algorithm instructions can be **simulated** by TMs.
- There are many other formalisms which are perfectly equivalent to TMs as for the class of computable functions they induce.
  - Examples: Random Access Machines, the λ-calculus, URMs, Partial Recursive Functions, ...

Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input x in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, f is said to be **computable** in time T.

Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input x in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, f is said to be **computable** in time T.
- A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time T iff f is computable in time T.

Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input x in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, f is said to be **computable** in time T.
- A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time T iff f is computable in time T.
- Examples.
  - The set of palindrome words is decidable in time $T(n) = 3n$.
  - Computing the parity of binary strings requires time $T(n) = n + 2$.
  - Basic operations like addition and multiplication are computable in polynomial time.

Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input x in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, f is said to be **computable** in time T.
- A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time T iff f is computable in time T.
- Examples.
  - The set of palindrome words is decidable in time $T(n) = 3n$.
  - Computing the parity of binary strings requires time $T(n) = n + 2$.
  - Basic operations like addition and multiplication are computable in polynomial time.
- A function $T : \mathbb{N} \to \mathbb{N}$ is **time-constructible** iff the function on $\{0,1\}^*$ defined as $x \mapsto \llcorner T(|x|) \lrcorner$ is computable.

On the Robustness of Our Definition

- **Question:** *why is our definition the right one?*

On the Robustness of Our Definition

- Question: *why is our definition the right one?*
- Actually, there are many details in our definion of a TM which are arbitrary: **many** alternative definitions are available in the literature.

On the Robustness of Our Definition

- Question: *why is our definition the right one?*
- Actually, there are many details in our definion of a TM which are arbitrary: **many** alternative definitions are available in the literature.
- Examples:
  - Rather than an arbitrary tape alphabet Γ, **restrict to** $\{0, 1, \triangleright, \Box\}$.
  - Just **one tape**, rather than many.
  - Tapes can be infinite **in both directions**.

On the Robustness of Our Definition

- Question: *why is our definition the right one?*
- Actually, there are many details in our definion of a TM which are arbitrary: **many** alternative definitions are available in the literature.
- Examples:
  - Rather than an arbitrary tape alphabet Γ, **restrict to** $\{0, 1, \triangleright, \square\}$.
  - Just **one tape**, rather than many.
  - Tapes can be infinite **in both directions**.
- In all the cases above (and in many others), one can prove that the more restrictive notion of machine **simulates** the more general one **with polynomial overhead**.
  - We do not have time to see all that, but you are encouraged to take a look at [AroraBarak2009], Section 1.3.1.

Machines as Strings

- One of the very nice consequences of keeping our definition of a Turing Machine very simple is that any machine $\mathcal{M} = (\Gamma, Q, \delta)$ is in fact copmletely determined from the graph of δ, seen as a subset of

$$Q \times \Gamma^k \times Q \times \Gamma^{k-1} \times \{\texttt{L}, \texttt{S}, \texttt{R}\}^k$$

Machines as Strings

- One of the very nice consequences of keeping our definition of a Turing Machine very simple is that any machine $\mathcal{M} = (\Gamma, Q, \delta)$ is in fact copmletely determined from the graph of δ, seen as a subset of

$$Q \times \Gamma^k \times Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$$

- Any parismomious encoding of δ as a binary string in $\{0, 1\}$ thus constitutes an acceptable encoding $\llcorner\mathcal{M}\lrcorner$ of $\mathcal{M}$, provided the following two conditions are satisfied:
    1. **Every string** in $\{0, 1\}^*$ represents a TM, i.e. for every $x \in \{0, 1\}^*$ there is $\mathcal{M}$ such that $x = \llcorner\mathcal{M}\lrcorner$.
    2. Every TM $\mathcal{M}$ is represented by an **infinitely many strings** (although exactly one is the "canonical" representation $\llcorner\mathcal{M}\lrcorner$ of $\mathcal{M}$.

Machines as Strings

- One of the very nice consequences of keeping our definition of a Turing Machine very simple is that any machine $\mathcal{M} = (\Gamma, Q, \delta)$ is in fact copmletely determined from the graph of δ, seen as a subset of

$$Q \times \Gamma^k \times Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$$

- Any parismomious encoding of δ as a binary string in $\{0, 1\}$ thus constitutes an acceptable encoding $\llcorner\mathcal{M}\lrcorner$ of $\mathcal{M}$, provided the following two conditions are satisfied:
    1. **Every string** in $\{0, 1\}^*$ represents a TM, i.e. for every $x \in \{0, 1\}^*$ there is $\mathcal{M}$ such that $x = \llcorner\mathcal{M}\lrcorner$.
    2. Every TM $\mathcal{M}$ is represented by an **infinitely many strings** (although exactly one is the "canonical" representation $\llcorner\mathcal{M}\lrcorner$ of $\mathcal{M}$.
- The two conditions above are not essential in any other contexts (i.e. when the encoded data is not a program), but are technically crucial here.

The Universal Turing Machine

Theorem (UTM, Efficiently)

There exists a TM $\mathcal{U}$ *such that for every* $x, \alpha \in \{0,1\}^*$, *it holds that* $\mathcal{U}(x,\alpha) = \mathcal{M}_\alpha(x)$, *where* $\mathcal{M}_\alpha$ *denotes the TM represented by* α. *Moreover, if* $\mathcal{M}_\alpha$ *halts on input* x *within* T *steps then* $\mathcal{U}(x,\alpha)$ *halts within* $CT\log(T)$ *steps, where* C *is independent of* $|x|$ *and depending only on* $\mathcal{M}_\alpha$.

The Universal Turing Machine

Theorem (UTM, Efficiently)

There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*$, it holds that $\mathcal{U}(x, \alpha) = \mathcal{M}_\alpha(x)$, where $\mathcal{M}_\alpha$ denotes the TM represented by α. Moreover, if $\mathcal{M}_\alpha$ halts on input x within T steps then $\mathcal{U}(x, \alpha)$ halts within $CT\log(T)$ steps, where C is independent of $|x|$ and depending only on $\mathcal{M}_\alpha$.

- A proof of the Theorem above in its full generality requires quite a bit of work.
- We will see the proof of a version of it in which the $O(T\log(T))$ bound is replaced by a polynomial one.

Uncomputability

Theorem (Uncomputable Functions Exist)

There exists a function $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

Uncomputability

Theorem (Uncomputable Functions Exist)

There exists a function $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

- The proof of the Theorem above is constructive. It suffices to consider the function

$$uc(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

Uncomputability

Theorem (Uncomputable Functions Exist)

There exists a function $uc : \{0,1\}^* \to \{0,1\}^*$ that is not computable by any Turing Machine.

- The proof of the Theorem above is constructive. It suffices to consider the function

$$uc(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

- Indeed, if uc were computable, there would be a TM $\mathcal{M}$ such that $\mathcal{M}(\alpha) = uc(\alpha)$ for every α, and in particular when $\alpha = \llcorner \mathcal{M} \lrcorner$.

Uncomputability

Theorem (Uncomputable Functions Exist)

There exists a function $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

- The proof of the Theorem above is constructive. It suffices to consider the function

$$uc(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

- Indeed, if uc were computable, there would be a TM $\mathcal{M}$ such that $\mathcal{M}(\alpha) = uc(\alpha)$ for every α, and in particular when $\alpha = \llcorner \mathcal{M} \lrcorner$.
- This would be a contraddiction, because by definition

$$uc(\llcorner \mathcal{M} \lrcorner) = 1 \Leftrightarrow \mathcal{M}(\llcorner \mathcal{M} \lrcorner) \neq 1$$

The Halting Problem

- The function uc we proved uncomputable does not represent an interesting computationl task.

The Halting Problem

- The function *uc* we proved uncomputable does not represent an interesting computationl task.
- Consider, instead, the function *halt* defined as follows:

$$halt(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{ halts on input } x; \\ 0 & \text{otherwise} \end{cases}$$

- Being able to compute *halt* would mean being able to check algorithms for termination.

The Halting Problem

- The function uc we proved uncomputable does not represent an interesting computationl task.
- Consider, instead, the function *halt* defined as follows:

$$halt(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{ halts on input } x; \\ 0 & \text{otherwise} \end{cases}$$

- Being able to compute *halt* would mean being able to check algorithms for termination.

Theorem (Uncomputability of *halt*)

The function halt is not computable by any TM.

The Halting Problem

- The function *uc* we proved uncomputable does not represent an interesting computationl task.
- Consider, instead, the function *halt* defined as follows:

$$halt(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{ halts on input } x; \\ 0 & \text{otherwise} \end{cases}$$

- Being able to compute *halt* would mean being able to check algorithms for termination.

Theorem (Uncomputability of *halt*)

The function halt is not computable by any TM.

- This result could be seen as a way to reinterpret Gödel's first incompleteness theorem "computationally".

Thank You!

Questions?