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 4: Reducibility

Table of Contents

Table of Contents

Reducibility

- So far, we have established the TM as our reference model of computation

Reducibility

- So far, we have established the TM as our reference model of computation
- We discuss what it means for a problem/function to be computed (either totally or partially)

Reducibility

- So far, we have established the TM as our reference model of computation
- We discuss what it means for a problem/function to be computed (either totally or partially)
- We also presented a typical problem – the halting problem - that is computationally undecidable

Reducibility

- So far, we have established the TM as our reference model of computation
- We discuss what it means for a problem/function to be computed (either totally or partially)
- We also presented a typical problem – the halting problem - that is computationally undecidable
- In the following, we will show how to prove that a problem is computationally undecidable by means of a specific technique called **reduction**

Reducibility

- Intuitively, a **reduction** is a way of translating one problem into another in such a way that the latter can be used to solve the former

Reducibility

- Intuitively, a **reduction** is a way of translating one problem into another in such a way that the latter can be used to solve the former
- Reduction is a pretty common practice in our every-day lives

Reducibility

- Intuitively, a **reduction** is a way of translating one problem into another in such a way that the latter can be used to solve the former
- Reduction is a pretty common practice in our every-day lives
- For example: Suppose that you want to find a way around a new city
  - You know that having a map of the city would solve your problem...

Reducibility

- Intuitively, a **reduction** is a way of translating one problem into another in such a way that the latter can be used to solve the former
- Reduction is a pretty common practice in our every-day lives
- For example: Suppose that you want to find a way around a new city
  - You know that having a map of the city would solve your problem...
  - Therefore, your original problem reduces to finding a map of the city!

Reducibility

- Reducibility always involves two problems: A and B

Reducibility

- Reducibility always involves two problems: A and B
- If A reduces to B, we can use a solution to B to solve A

Reducibility

- Reducibility always involves two problems: A and B
- If A reduces to B, we can use a solution to B to solve A
- In our example above:
  - A is the original problem of finding a way around a new city

Reducibility

- Reducibility always involves two problems: A and B
- If A reduces to B, we can use a solution to B to solve A
- In our example above:
  - A is the original problem of finding a way around a new city
  - B is the problem of finding a map

Reducibility

- Reducibility always involves two problems: A and B
- If A reduces to B, we can use a solution to B to solve A
- In our example above:
  - A is the original problem of finding a way around a new city
  - B is the problem of finding a map

Note

Reducibility does **not** say anything about solving A or B, but just about the solvability of A in the presence of a solution to B

Reducibility

Reducibility occurs often in mathematical problems:

Examples

The problem of measuring the area of a rectangle (A) reduces to the problem of finding the size of its length and width (B).

Reducibility

Reducibility occurs often in mathematical problems:

Examples

The problem of measuring the area of a rectangle (A) reduces to the problem of finding the size of its length and width (B).
The problem of solving a system of linear equations (A) reduces to inverting the matrix of coefficients (B).

Reducibility

- Reducibility plays a crucial role in classifying problems according to their (un)decidability

Reducibility

- Reducibility plays a crucial role in classifying problems according to their (un)decidability
- Actually, it allows us to also further classify the set of decidable problems into classes of complexity (more on this later)

Reducibility

- Reducibility plays a crucial role in classifying problems according to their (un)decidability
- Actually, it allows us to also further classify the set of decidable problems into classes of complexity (more on this later)
- When A is reducible to B (often denoted as $A \leq B$) it means that solving A cannot be harder than solving B (because a solution to B gives a solution to A)

Reducibility

- Reducibility plays a crucial role in classifying problems according to their (un)decidability
- Actually, it allows us to also further classify the set of decidable problems into classes of complexity (more on this later)
- When A is reducible to B (often denoted as $A \leq B$) it means that solving A cannot be harder than solving B (because a solution to B gives a solution to A)

Corollary

- *If $A \leq B$ and B is* **decidable** *then A is also* **decidable**

Reducibility

- Reducibility plays a crucial role in classifying problems according to their (un)decidability
- Actually, it allows us to also further classify the set of decidable problems into classes of complexity (more on this later)
- When A is reducible to B (often denoted as $A \leq B$) it means that solving A cannot be harder than solving B (because a solution to B gives a solution to A)

Corollary

- *If $A \leq B$ and B is* **decidable** *then A is also* **decidable**
- *If $A \leq B$ and A is* **undecidable** *then B is also* **undecidable**

Reducibility

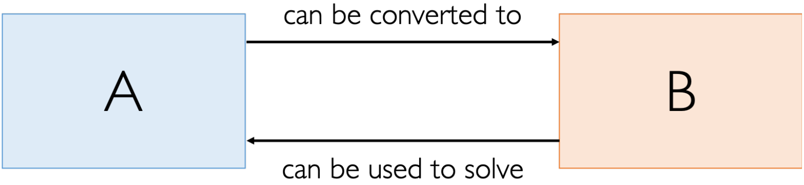


Table of Contents

The (Actual) Halting Problem

- We have already proved that $HALT_{ACC}$ (equivalently, A_{TM}) is **undecidable**

The (Actual) Halting Problem

- We have already proved that $HALT_{ACC}$ (equivalently, A_{TM}) is **undecidable**
- We call this "the halting problem": the problem of determining if a TM halts **and** accepts a given input

The (Actual) Halting Problem

- We have already proved that $HALT_{ACC}$ (equivalently, A_{TM}) is **undecidable**
- We call this "the halting problem": the problem of determining if a TM halts **and** accepts a given input
- In fact, the *actual* halting problem is subtly different from that!

The (Actual) Halting Problem

- We have already proved that $HALT_{ACC}$ (equivalently, A_{TM}) is **undecidable**
- We call this "the halting problem": the problem of determining if a TM halts **and** accepts a given input
- In fact, the *actual* halting problem is subtly different from that!
- We refer to $HALT$ as the problem of determining if a TM halts (either accepting or rejecting) a given input

$$HALT(\langle M, x\rangle) = \begin{cases} 1 & \text{if } M(x) = 1 \vee M(x) = 0 \\ 0 & M(x) = \perp \end{cases}$$

The (Actual) Halting Problem

Theorem

The function HALT is **not** *total and computable. Analogously, the language it defines* $H_{TM} = \{\langle M, x \rangle \mid M \text{ is a TM}, M(x) = 1 \vee M(x) = 0\}$ *is* **undedicidable**

The (Actual) Halting Problem

Theorem

The function HALT is **not** *total and computable. Analogously, the language it defines* $H_{TM} = \{\langle M, x\rangle \mid M \text{ is a TM}, M(x) = 1 \vee M(x) = 0\}$ *is* **undedicidable**

Note

HALT is of course **partially** computable, i.e., H_{TM} is **recognizable**.
A recognizer for H_{TM} is a TM that works as follows: it just runs $M(x)$ and if this ever halts (either accepting or rejecting) will output 1, otherwise it will loop forever

The (Actual) Halting Problem is Undecidable

- To prove the theorem above, we again look for a contradiction

The (Actual) Halting Problem is Undecidable

- To prove the theorem above, we again look for a contradiction
- We assume that H_{TM} is decidable and we use that assumption to show that A_{TM} would be decidable as well

The (Actual) Halting Problem is Undecidable

- To prove the theorem above, we again look for a contradiction
- We assume that H_{TM} is decidable and we use that assumption to show that A_{TM} would be decidable as well
- The idea is to reduce a problem A which we know is undecidable (A_{TM}) to another problem B which we assume to be decidable (H_{TM}) and get to a contradiction

The (Actual) Halting Problem is Undecidable: Proof

- Let's assume we have a **decider** M_1 for H_{TM}, i.e., a TM that decides H_{TM}

The (Actual) Halting Problem is Undecidable: Proof

- Let's assume we have a **decider** M_1 for H_{TM}, i.e., a TM that decides H_{TM}
- If that is the case, we can use M_1 to construct another **decider** M_2, which decides A_{TM}

The (Actual) Halting Problem is Undecidable: Proof

- Let's assume we have a **decider** M_1 for H_{TM}, i.e., a TM that decides H_{TM}
- If that is the case, we can use M_1 to construct another **decider** M_2, which decides A_{TM}
- To get the idea of how to build M_2, pretend you are M_2: your task is to decide A_{TM}

The (Actual) Halting Problem is Undecidable: Proof

- Let's assume we have a **decider** M_1 for H_{TM}, i.e., a TM that decides H_{TM}
- If that is the case, we can use M_1 to construct another **decider** M_2, which decides A_{TM}
- To get the idea of how to build M_2, pretend you are M_2: your task is to decide A_{TM}
- M_2 takes an input in the form of $\langle M, x\rangle$ and must output 1 if M halts and accepts x, or 0 if M either halts and rejects or loop forever on x

The (Actual) Halting Problem is Undecidable: Proof

- Let's assume we have a **decider** M_1 for H_{TM}, i.e., a TM that decides H_{TM}
- If that is the case, we can use M_1 to construct another **decider** M_2, which decides A_{TM}
- To get the idea of how to build M_2, pretend you are M_2: your task is to decide A_{TM}
- M_2 takes an input in the form of $\langle M, x \rangle$ and must output 1 if M halts and accepts x, or 0 if M either halts and rejects or loop forever on x
- The problem for M_2, of course, is it can't loop forever (it is a decider!)

The (Actual) Halting Problem is Undecidable: Proof

- How could M_2 take advantage of the assumption of the existence of a decider M_1 for H_{TM}?

The (Actual) Halting Problem is Undecidable: Proof

- How could M_2 take advantage of the assumption of the existence of a decider M_1 for H_{TM}?
- M_2 could first run M_1 on $\langle M, x \rangle$ to test if M halts on x:
  - If M_1 indicates that M does not halt on x, M_2 can output 0

The (Actual) Halting Problem is Undecidable: Proof

- How could M_2 take advantage of the assumption of the existence of a decider M_1 for H_{TM}?
- M_2 could first run M_1 on $\langle M, x\rangle$ to test if M halts on x:
  - If M_1 indicates that M does not halt on x, M_2 can output 0
  - If M_1, instead, indicates that M halts on x, M_2 can run safely M on x and will output whatever M will (1 if M accepts, 0 if it rejects)

The (Actual) Halting Problem is Undecidable: Proof

- How could M_2 take advantage of the assumption of the existence of a decider M_1 for H_{TM}?
- M_2 could first run M_1 on $\langle M, x \rangle$ to test if M halts on x:
  - If M_1 indicates that M does not halt on x, M_2 can output 0
  - If M_1, instead, indicates that M halts on x, M_2 can run safely M on x and will output whatever M will (1 if M accepts, 0 if it rejects)
- Thus, if M_1 exits, we can use it to decide A_{TM}

The (Actual) Halting Problem is Undecidable: Proof

- How could M_2 take advantage of the assumption of the existence of a decider M_1 for H_{TM}?
- M_2 could first run M_1 on $\langle M, x \rangle$ to test if M halts on x:
  - If M_1 indicates that M does not halt on x, M_2 can output 0
  - If M_1, instead, indicates that M halts on x, M_2 can run safely M on x and will output whatever M will (1 if M accepts, 0 if it rejects)
- Thus, if M_1 exits, we can use it to decide A_{TM}
- Since we know that A_{TM} is undecidable, M_1 cannot exist and therefore H_{TM} is undecidable as well

The (Actual) Halting Problem is Undecidable: Proof

H_{TM} is undecidable.

Let's assume for the purpose of obtaining a contradiction that a TM M_1 exists and decides H_{TM}. We construct *another* TM M_2 that decides A_{TM}, therefore getting to a contradiction.

$M_2 =$"On input $\langle M, x\rangle$, i.e., an encoding of a TM M and a string x:

1. Run TM M_1 on input $\langle M, x\rangle$;

The (Actual) Halting Problem is Undecidable: Proof

H_{TM} is undecidable.

Let's assume for the purpose of obtaining a contradiction that a TM M_1 exists and decides H_{TM}. We construct *another* TM M_2 that decides A_{TM}, therefore getting to a contradiction.

$M_2 =$"On input $\langle M, x \rangle$, i.e., an encoding of a TM M and a string x:

1. Run TM M_1 on input $\langle M, x \rangle$;
2. If M_1 returns 0, it means M does not halt on x, therefore M_2 returns 0 as well;

The (Actual) Halting Problem is Undecidable: Proof

H_{TM} is undecidable.

Let's assume for the purpose of obtaining a contradiction that a TM M_1 exists and decides H_{TM}. We construct *another* TM M_2 that decides A_{TM}, therefore getting to a contradiction.

$M_2 =$"On input $\langle M, x \rangle$, i.e., an encoding of a TM M and a string x:

1. Run TM M_1 on input $\langle M, x \rangle$;
2. If M_1 returns 0, it means M does not halt on x, therefore M_2 returns 0 as well;
3. If M_1 returns 1, it means M halts on x, therefore M_2 can safely simulate M on x until it halts;

The (Actual) Halting Problem is Undecidable: Proof

H_{TM} is undecidable.

Let's assume for the purpose of obtaining a contradiction that a TM M_1 exists and decides H_{TM}. We construct *another* TM M_2 that decides A_{TM}, therefore getting to a contradiction.

M_2 ="On input $\langle M, x\rangle$, i.e., an encoding of a TM M and a string x:

1. Run TM M_1 on input $\langle M, x\rangle$;
2. If M_1 returns 0, it means M does not halt on x, therefore M_2 returns 0 as well;
3. If M_1 returns 1, it means M halts on x, therefore M_2 can safely simulate M on x until it halts;
4. If M halts and accepts, M_2 returns 1; otherwise (M halts and rejects), M_2 returns 0."

□

Table of Contents

Mapping Reducibility

- We have shown how to use the reducibility technique to prove that a problem is undecidable

Mapping Reducibility

- We have shown how to use the reducibility technique to prove that a problem is undecidable
- Now, we formalize the notion of reducibility which will be used later on in the context of complexity theory

Mapping Reducibility

- We have shown how to use the reducibility technique to prove that a problem is undecidable
- Now, we formalize the notion of reducibility which will be used later on in the context of complexity theory
- We choose to define reducibility according to the definition of **mapping reducibility** (a.k.a. **many-to-one reducibility**)

Mapping Reducibility

- Roughly speaking, being able to reduce problem A to problem B using mapping reducibility means that there exists a **total computable function** converting instances of problem A to instances of problem B

Mapping Reducibility

- Roughly speaking, being able to reduce problem A to problem B using mapping reducibility means that there exists a **total computable function** converting instances of problem A to instances of problem B
- If we have such a mapping function – called **reduction** – we can solve probelm A using a solver for problem B

Mapping Reducibility

- Roughly speaking, being able to reduce problem A to problem B using mapping reducibility means that there exists a **total computable function** converting instances of problem A to instances of problem B
- If we have such a mapping function – called **reduction** – we can solve probelm A using a solver for problem B
- The reason is that any instance of A can be solved by:
  1. Using the reduction to convert it to an instance of B;
  2. Applying the solver for B

Computable Functions (Again)

A TM computes a function by starting with the input (to that function) on its tape and halting with the output (of that function) on its tape

Computable Functions (Again)

A TM computes a function by starting with the input (to that function) on its tape and halting with the output (of that function) on its tape

Definition

A function $f : \Sigma^* \mapsto \Sigma^*$ is a **total computable function** if some Turing machine M, on **every** input x, **halts and outputs** $f(x)$

Computable Functions: Examples

- All usual arithmetic operations on integers are total computable functions. For example, $add(m, n) = m + n$

Computable Functions: Examples

- All usual arithmetic operations on integers are total computable functions. For example, $add(m, n) = m + n$
- Transformations of TM encodings are total computable functions. For example, the function f that takes as input $x = \langle M \rangle$ and outputs $f(x) = \langle M' \rangle$

Computable Functions: Examples

- All usual arithmetic operations on integers are total computable functions. For example, $add(m, n) = m + n$
- Transformations of TM encodings are total computable functions. For example, the function f that takes as input $x = \langle M \rangle$ and outputs $f(x) = \langle M' \rangle$

Note

We can always build a TM M' starting from (the encoding of) another TM M in a finite and definite amount of steps. This is because every TM is fully specified by its **transition function**, which is finite!

Computable Functions: Examples

- Anything that a TM can do without looping, including running deciders, is permissible

Computable Functions: Examples

- Anything that a TM can do without looping, including running deciders, is permissible
- If the form of the input is wrong (e.g., if the TM is expecting $\langle M, x \rangle$ but gets something else), then it clears the tape and halts (i.e., outputs ϵ)

Mapping Reducibility: Formal Definition

Definition

A problem (language) A is **mapping reducibile** to problem (language) B, i.e., $A \leq_m B$, if there exists a **total computable function** $f : \Sigma^* \mapsto \Sigma^*$, where **for every** $x \in A$ it holds:

$$x \in A \Longleftrightarrow f(x) \in B$$

The function f is called a **reduction** of A to B

Mapping Reducibility: Formal Definition

Definition

A problem (language) A is **mapping reducibile** to problem (language) B, i.e., $A \leq_m B$, if there exists a **total computable function** $f : \Sigma^* \mapsto \Sigma^*$, where **for every** $x \in A$ it holds:

$$x \in A \iff f(x) \in B$$

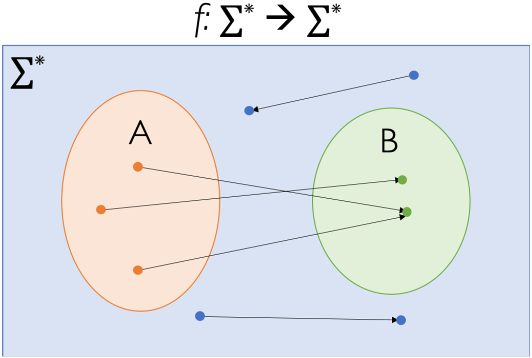
The function f is called a **reduction** of A to B

Note

To prove that f is a mapping reduction we need to show that:

- $x \in A \implies f(x) = x' \in B$: f maps elements of A to elements of B
- $x \notin A \implies f(x) = x' \notin B$: f maps element of $\overline{A}$ to elements of $\overline{B}$

Mapping Reducibility: Formal Definition



Mapping Reducibility: Formal Definition

- A mapping reduction of A to B provides a way to translate questions about membership testing in A to membership testing in B

Mapping Reducibility: Formal Definition

- A mapping reduction of A to B provides a way to translate questions about membership testing in A to membership testing in B
- To test if $x \in A$, we use the reduction f to map x to $f(x)$ and test whether $f(x) \in B$

Mapping Reducibility: Formal Definition

- A mapping reduction of A to B provides a way to translate questions about membership testing in A to membership testing in B
- To test if $x \in A$, we use the reduction f to map x to $f(x)$ and test whether $f(x) \in B$
- Of course, if one problem is mapping-reducible to another, previously solved, problem, we can therefore obtain a solution to the original problem

Mapping Reducibility: Consequences

Theorem

If $A \leq_m B$ and B is **decidable**, *then A is also* **decidable**

Mapping Reducibility: Consequences

Theorem

If $A \leq_m B$ and B is **decidable**, *then A is also* **decidable**

Proof.

We let M_B be the decider for B and f the reduction from A to B. We therefore describe a decider M_A for A as follows:

$M_A =$ "On input x:

1. Compute $f(x)$;
2. Run M_B on $f(x)$ and if M_B accepts then **accepts**; otherwise **rejects**."

Mapping Reducibility: Consequences

Theorem

If $A \leq_m B$ and B is **decidable**, *then A is also* **decidable**

Proof.

We let M_B be the decider for B and f the reduction from A to B. We therefore describe a decider M_A for A as follows:

$M_A =$ "On input x:

1. Compute $f(x)$;
2. Run M_B on $f(x)$ and if M_B accepts then **accepts**; otherwise **rejects**."

If $x \in A$ then $f(x) \in B$ (because f is a reduction from A to B), M_B and thus M_A **accept**.

If $x \notin A$ then $f(x) \notin B$, M_B and thus M_A **reject**.

M_A is a decider for A $\square$

Mapping Reducibility: Consequences

Corollary

If $A \leq_m B$ and A is **undecidable** *then B is also* **undecidable**

Mapping Reducibility: Consequences

Corollary

If $A \leq_m B$ and A is **undecidable** *then B is also* **undecidable**

Proof.

We build a TM M_f that computes the mapping reduction f as follows:

$M_f =$ "On input ⟨an instance of problem A⟩:

1. Construct an instance of problem B;
2. Output ⟨the instance of problem B⟩"

□

Mapping Reducibility: Consequences

Corollary

If $A \leq_m B$ and A is **undecidable** *then B is also* **undecidable**

Proof.

We build a TM M_f that computes the mapping reduction f as follows:
$M_f =$ "On input $\langle$an instance of problem $A\rangle$:

1. Construct an instance of problem B;
2. Output $\langle$the instance of problem $B\rangle$"

$\square$

Note

Rather than accept or reject, the TM M_f corresponding to the mapping outputs the result of the reduction

Mapping Reducibility: Consequences

Corollary

If $A \leq_m B$ and B is **recognizable** *then A is also* **recognizable**

Mapping Reducibility: Consequences

Corollary

If $A \leq_m B$ and B is **recognizable** *then A is also* **recognizable**

Corollary

If $A \leq_m B$ and A is **unrecognizable** *then B is also* **unrecognizable**

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

Example

We have already used a reduction from A_{TM} to H_{TM} to prove the latter is **undecidable**. In particular, this reduction shows how a decider for H_{TM} could be used to build a decider also for A_{TM}, which we know is, in fact, undecidable.

We want to find a reduction f that takes as input $\langle M, x\rangle$ and transforms it into $\langle M', x\rangle$, such that:

$$\langle M, x\rangle \in A_{TM} \iff \langle M', x\rangle \in H_{TM}$$

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

The mapping reduction f can be computed by the following TM M_f

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

The mapping reduction f can be computed by the following TM M_f
$M_f =$ "On input $\langle M, x \rangle$:

1. Construct the following **new** TM M':
   $M' =$ "On input z:
   1. Call M on z;
   2. If $M(z) = 1$ (i.e., if M accepts z), then $M'(z) = 1$ as well;
   3. If $M(z) = 0$ (i.e., if M rejects z), then $M'(z)$ enters in a loop"
2. Finally, output $\langle M', x \rangle$"

Note

If $M(z) = \perp$ $M'(z) = \perp$ as well, and there is no need of doing anything!

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

The mapping reduction f can be computed by the following TM M_f

$M_f =$ "On input $\langle M, x \rangle$:

1. Construct the following **new** TM M':
   $M' =$ "On input z:
   1. Call M on z;
   2. If $M(z) = 1$ (i.e., if M accepts z), then $M'(z) = 1$ as well;
   3. If $M(z) = 0$ (i.e., if M rejects z), then $M'(z)$ enters in a loop;
2. Finally, output $\langle M', x \rangle$."

Note

Building the TM M' can be done in a finite number of steps (i.e., does not loop) so M_f can't loop either. M' itself may not halt, in fact it will not on some inputs, but the point is we can build its (finite) representation using a total computable function f implemented by M_f.

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

The mapping reduction f can be computed by the following TM M_f
M_f = "On input $\langle M, x\rangle$:

1. Construct the following **new** TM M':
   M' = "On input z:
   1. Call M on z;
   2. If $M(z) = 1$ (i.e., if M accepts z), then $M'(z) = 1$ as well;
   3. If $M(z) = 0$ (i.e., if M rejects z), then $M'(z)$ enters in a loop;
2. Finally, output $\langle M', x\rangle$."

Note

If $\langle M, x\rangle \in A_{TM}$, then M accepts x so does M' and thus halts on x, i.e., $\langle M', x\rangle \in H_{TM}$

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

The mapping reduction f can be computed by the following TM M_f

$M_f =$ "On input $\langle M, x \rangle$:

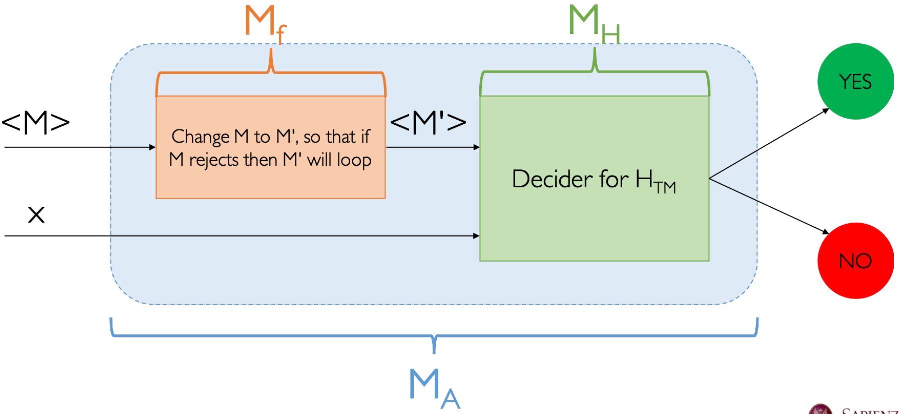
1. Construct the following **new** TM M':
   $M' =$ "On input z:
   1. Call M on z;
   2. If $M(z) = 1$ (i.e., if M accepts z), then $M'(z) = 1$ as well;
   3. If $M(z) = 0$ (i.e., if M rejects z), then $M'(z)$ enters in a loop;
2. Finally, output $\langle M', x \rangle$."

Note

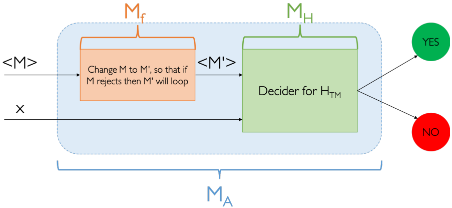
If $\langle M, x \rangle \in A_{TM}$, then M accepts x so does M' and thus halts on x, i.e., $\langle M', x \rangle \in H_{TM}$

If $\langle M, x \rangle \notin A_{TM}$, then M rejects or loops on x and in either case M' loops on x, i.e., $\langle M', x \rangle \notin H_{TM}$

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$



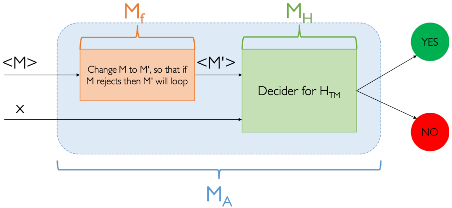
Mapping Reducibility: $A_{TM} \leq_m H_{TM}$



Upon receiving its input $\langle M, x\rangle$, the TM M_A works as follows:

1. It computes the reduction f by running the TM M_f, which transforms $\langle M, x\rangle$ into $f(\langle M, x\rangle) = \langle M', x\rangle$, where M' loops whenever M rejects;

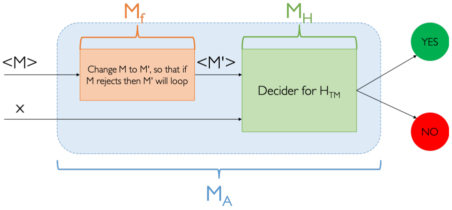
Mapping Reducibility: $A_{TM} \leq_m H_{TM}$



Upon receiving its input $\langle M, x \rangle$, the TM M_A works as follows:

1. It computes the reduction f by running the TM M_f, which transforms $\langle M, x \rangle$ into $f(\langle M, x \rangle) = \langle M', x \rangle$, where M' loops whenever M rejects;
2. It runs M_H on $\langle M', x \rangle$;

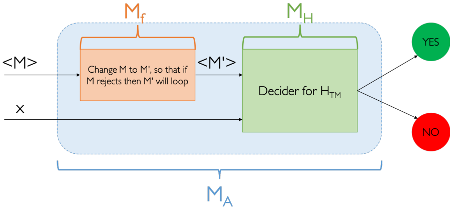
Mapping Reducibility: $A_{TM} \leq_m H_{TM}$



Upon receiving its input $\langle M, x\rangle$, the TM M_A works as follows:

1. It computes the reduction f by running the TM M_f, which transforms $\langle M, x\rangle$ into $f(\langle M, x\rangle) = \langle M', x\rangle$, where M' loops whenever M rejects;
2. It runs M_H on $\langle M', x\rangle$;
3. If M_H outputs 1 (i.e., **accepts**), then M_A outputs 1;

Mapping Reducibility: $A_{TM} \leq_m H_{TM}$

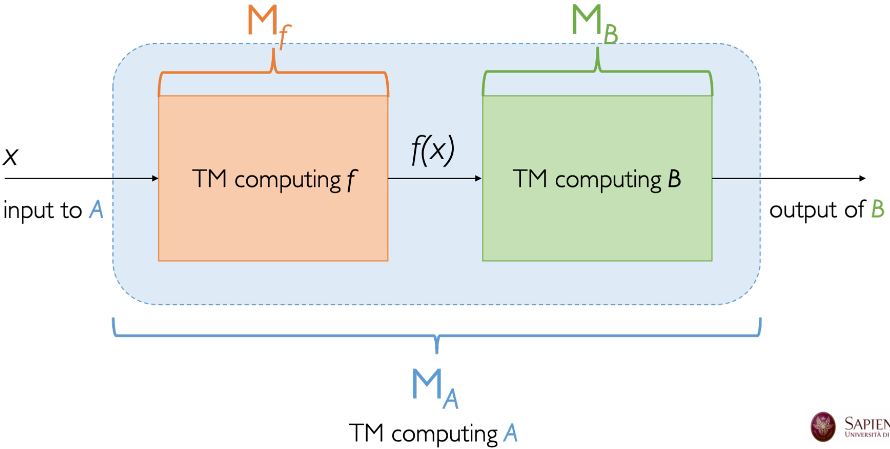


Upon receiving its input $\langle M, x\rangle$, the TM M_A works as follows:

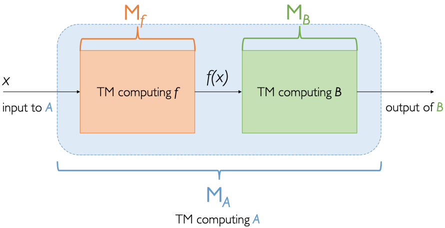
1. It computes the reduction f by running the TM M_f, which transforms $\langle M, x\rangle$ into $f(\langle M, x\rangle) = \langle M', x\rangle$, where M' loops whenever M rejects;
2. It runs M_H on $\langle M', x\rangle$;
3. If M_H outputs 1 (i.e., **accepts**), then M_A outputs 1;
4. If M_H outputs 0 (i.e., **rejects**), then M_A outputs 0;

Mapping Reducibility: General Pattern

$$A \leq_m B$$



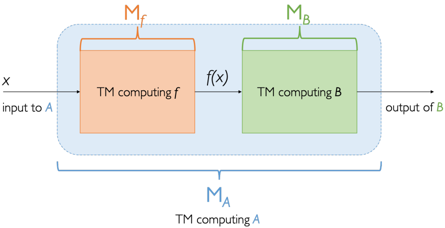
Mapping Reducibility: General Pattern



Upon receiving x, i.e., an input instance of problem A, the TM M_A works as follows:

1. It transforms $x \in A$ into $f(x) \in B$, using M_f;

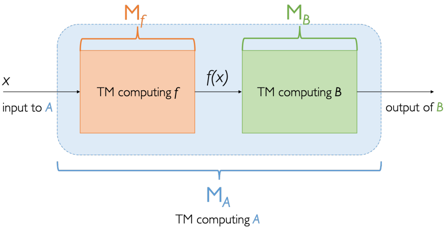
Mapping Reducibility: General Pattern



Upon receiving x, i.e., an input instance of problem A, the TM M_A works as follows:

1. It transforms $x \in A$ into $f(x) \in B$, using M_f;
2. It runs M_B (solver of problem B) on $f(x)$;

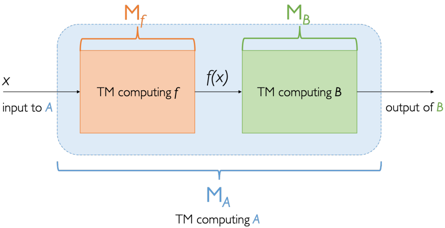
Mapping Reducibility: General Pattern



Upon receiving x, i.e., an input instance of problem A, the TM M_A works as follows:

1. It transforms $x \in A$ into $f(x) \in B$, using M_f;
2. It runs M_B (solver of problem B) on $f(x)$;
3. If M_B outputs 1 (i.e., **accepts**), then M_A outputs 1;

Mapping Reducibility: General Pattern



Upon receiving x, i.e., an input instance of problem A, the TM M_A works as follows:

1. It transforms $x \in A$ into $f(x) \in B$, using M_f;
2. It runs M_B (solver of problem B) on $f(x)$;
3. If M_B outputs 1 (i.e., **accepts**), then M_A outputs 1;
4. If M_B outputs 0 (i.e., **rejects**), then M_A outputs 0;