Turing reductions

The reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either.

(Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.)

Turing reductions

The reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either.

(Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.)

A reduction to show that problem P is not semi-decidable works analogously: Assume that a semi-algorithm for P exists. Build a semi-algorithm A' (using A as a subroutine) for a problem P' that is known to be non-semi-decidable. As such A' cannot exist, semi-algorithm A cannot exist either.

Turing reductions

The reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either.

(Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.)

A reduction to show that problem P is not semi-decidable works analogously: Assume that a semi-algorithm for P exists. Build a semi-algorithm A' (using A as a subroutine) for a problem P' that is known to be non-semi-decidable. As such A' cannot exist, semi-algorithm A cannot exist either.

Such reductions are called **Turing reductions**. (Later we'll see some other, weaker types of reductions.)

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.
2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.
2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Note that

- **if M accepts w then M' accepts all words**,
- **if M does not accept w then M' does not accept any word**.

So A returns "yes" on input M' if and only if M accepts w.

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.
2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Note that

- **if M accepts w then M' accepts all words**,
- **if M does not accept w then M' does not accept any word**.

So A returns "yes" on input M' if and only if M accepts w.

Algorithm A' described above cannot exist (solves a known undecidable problem) so the hypothetical algorithm A does not exist.

A universal Turing machine

Recall

$$L_u = \{\ \langle M \rangle \# w \mid \ M \text{ accepts the input word } w\ \}$$

We saw that L_u is not recursive.

However, L_u is recursively enumerable. (Informally: the semi-algorithm first checks that the given input is of the correct form $\langle M \rangle \# w$. It then simulates the Turing machine M on the input w until – if ever – M halts. If M halts in the final state then semi-algorithm answers "yes".)

A universal Turing machine

Recall

$$L_u = \{\ \langle M \rangle \# w \mid \ M \text{ accepts the input word } w\ \}$$

We saw that L_u is not recursive.

However, L_u is recursively enumerable. (Informally: the semi-algorithm first checks that the given input is of the correct form $\langle M \rangle \# w$. It then simulates the Turing machine M on the input w until – if ever – M halts. If M halts in the final state then semi-algorithm answers "yes".)

Recursively enumerable languages are recognized by Turing machines, so there exists a Turing machine M_u that recognizes L_u. Such a machine M_u is called a **universal Turing machine**. It can simulate any given Turing machine M on any given input w, when given the description (=encoding $\langle M \rangle$) of the machine to be simulated.

So M_u is a **programmable computer**: rather than building a new TM for each new language, one can use the same TM M_u and only change the "program" $\langle M \rangle$ that describes which Turing machine should be simulated.

We can also make the following observations:

Corollary. There are recursively enumerable languages that are not recursive.

Proof. For example L_u is such a language, as is also L_{halt} and the complement of L_d.

We can also make the following observations:

Corollary. There are recursively enumerable languages that are not recursive.

Proof. For example L_u is such a language, as is also L_{halt} and the complement of L_d.

Corollary. The family of recursively enumerable languages is not closed under complementation.

Proof. Any language L that is recursively enumerable but not not recursive confirms this.

(If L and the complement of L would be recursively enumerable then L would be recursive.)

A summary of the characters of the play so far:

- $\langle M \rangle$ denotes the encoding (as a word over the alphabet $\{a, b\}$) of a Turing machine M having the input alphabet $\{a, b\}$.

- $L_{enc} = \{\langle M \rangle \mid \ M \text{ is a Turing machine } \}$ is the language containing all valid encodings of such Turing machines. It is recursive.

- $L_d = \{\langle M \rangle \mid \langle M \rangle \notin L(M)\}$ is the complemented diagonal language that contains encodings $\langle M \rangle$ of Turing machines M that do not accept their own encodings $\langle M \rangle$. It is not recursively enumerable.

- $L_{halt} = \{\langle M \rangle \mid \ \text{Turing machine } M \text{ halts on the blank initial tape } \}$. It is r.e. but not recursive.

- $L_u = \{\langle M \rangle \# w \mid w \in L(M)\}$ is the language containing encodings of Turing machines and inputs that they accept. It is also r.e. but not recursive.

- M_u is a universal Turing machine that recognizes the language L_u, *i.e.*, $L(M_u) = L_u$.

Rice's theorem

We have seen many undecidable questions that concern Turing machines. Some questions are clearly decidable (e.g., "Does a given Turing machine have 5 states ?").

Rice's theorem states that any non-trivial question that only concerns the language that a TM recognizes, rather than the machine itself, is undecidable.

Rice's theorem

We have seen many undecidable questions that concern Turing machines. Some questions are clearly decidable (e.g., "Does a given Turing machine have 5 states ?").

Rice's theorem states that any non-trivial question that only concerns the language that a TM recognizes, rather than the machine itself, is undecidable.

More precisely: Let $\mathcal{P}$ be any family of languages. We call $\mathcal{P}$ a **non-trivial property** of Turing machines if there exist Turing machines M_1 and M_2 such that $L(M_1) \in \mathcal{P}$ and $L(M_2) \notin \mathcal{P}$.

Theorem. Let $\mathcal{P}$ be a non-trivial property of Turing machines. There is no algorithm to determine if a given Turing machine M has $L(M) \in \mathcal{P}$.

Proof. On the blackboard.

Example. Rice's theorem shows, for example, that the following questions are undecidable:

- "Does a given TM accept all words in $\{a, b\}^*$?"
- "Is $L(M)$ regular for a given TM M ?"
- "Does a given Turing machine accept all palindromes in $\{a, b\}^*$?"
- ...

Example. Rice's theorem shows, for example, that the following questions are undecidable:

- "Does a given TM accept all input words ?"
- "Is $L(M)$ regular for a given TM M ?"
- "Does a given Turing machine accept all palindromes ?"
- ...

Remark. A careful analysis of the proof of Rice's theorem shows that, for a non-trivial property $\mathcal{P}$ such that $\emptyset \in \mathcal{P}$, it is non-semi-decidable for a given Turing machine M whether $L(M) \in \mathcal{P}$.

Other undecidable problems: rewrite systems

A **semi-Thue system** $T = (\Sigma, R)$ consists of

- an alphabet Σ, and
- a finite set R of **rewrite rules**

$$u \longrightarrow v$$

where $u, v \in \Sigma^*$.

Other undecidable problems: rewrite systems

A **semi-Thue system** $T = (\Sigma, R)$ consists of

- an alphabet Σ, and
- a finite set R of **rewrite rules**

$$u \longrightarrow v$$

where $u, v \in \Sigma^*$.

A rewrite rule $u \longrightarrow v$ allows **derivation steps**

$$x\ u\ y \Rightarrow x\ v\ y$$

for all $x, y \in \Sigma^*$.

In other words, a derivation step consists of replacing subword u by v.

Notations $x \Rightarrow^* y$ and $x \Rightarrow^+ y$ and $x \Rightarrow^i y$ have the usual interpretation.

Example. Let $T = (\{a, b\}, R)$ where R contains three rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

Then $ababa \Rightarrow^* bab$ because

$$\underline{aba}ba \Rightarrow ba\underline{bb}a \Rightarrow b\underline{aba} \Rightarrow \underline{bb}ab \Rightarrow bab$$

(Underlining indicates the subword being rewritten.)

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$aba \Rightarrow$

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$\underline{a}ba \Rightarrow a\underline{aba} \Rightarrow ab\underline{a}b \Rightarrow \underline{aba}ab \Rightarrow babab$

Yes.

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$\underline{a}ba \Rightarrow a\underline{aba} \Rightarrow ab\underline{a}b \Rightarrow \underline{aba}ab \Rightarrow babab$

Yes.

What about a derivation $babab \Rightarrow^* aba$ in the opposite direction ?

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$\underline{a}ba \Rightarrow \underline{aaba} \Rightarrow ab\underline{a}b \Rightarrow \underline{aba}ab \Rightarrow babab$

Yes.

What about a derivation $babab \Rightarrow^* aba$ in the opposite direction ?

No. If a word begins with the letter b then all derived words also begin with the letter b.

In general, it is difficult to solve the word problem. The problem is undecidable.

In general, it is difficult to solve the word problem. The problem is undecidable.

But we prove more. The **individual word problem** associated to a fixed semi-Thue system T and a fixed target word y asks:

"Does a given word x derive y in T ?"

The difference to the word problem is that T and x are not part of the input, but are rather **fixed**. Different choices of T and y induce different decision problems. For some T and y the problem is decidable, but we prove in the following the for a suitable choice of T and y the problem is undecidable:

Theorem. There exists a semi-Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

(Note that it immediately follows that the word problem is undecidable as well.)