LIMITS OF COMPUTABILITY

Decidable problems

- Recall: A decision problem is ***decidable*** if there is a decision algorithm.
- We can now make this more precise.
- A language $L \subseteq \Sigma^*$ is ***(Turing-) decidable***
  if it is recognized by some ***Turing-decider,*** that is
  a Turing acceptor that ***terminates for every input***.
- A decision problem is Turing-decidable if
  its textual representation is.
- Given the Turing-Church Thesis we
  identify informal algorithms with Turing acceptors!

Decision problems about computing devices

- **ε-ACCEPTANCE**:

  Given acceptor M *does* M *accept* ε*?*

- **NON-EMPTINESS**:

  Given acceptor M*, does it accept some string?*

- **TOTALITY**:

  Given acceptor M *does* M *accept every string?*

- **ACCEPTANCE**:

  Given acceptor M *and string* w*, does* M *accept* w*?*

- **HALTING**:

  Given a Turing transducer M *and string* w*, does* M *terminate on input* w *?.*

Decidability preseved under set operations

- Let $\mathcal{P}$ and $\mathcal{Q}$ be problems referring to the same instances, decided by algorithms A_P and A_Q respectively.
- The ***complement*** of $\mathcal{P}$ is decidable:

  to decide $w \in \bar{\mathcal{P}}$ run A_P on input w

  and flip the answer.
- The ***intersection*** of $\mathcal{P}$ and $\mathcal{Q}$ is decidable:

  to decide $w \in \mathcal{P} \cap \mathcal{Q}$

  - Run A_P on w, if it rejects, reject; if it accepts:
  - run A_Q on w, if it rejects, reject; if it accepts accept.
- The union of the problems is also decidable, by a similar argument.

REDUCTIONS BETWEEN PROBLEMS

Using other problems’s solution

- We often fulfill tasks using tools developed for other tasks.
- Examples:

1. To match two decks of card, first sort them.
2. To use biased coins when a fair coin is needed

    use a biased coin in double-rounds:

    take HT as “head,” TH as “tail,” discard HH and TT.
3. Use calculator whose multiplication works only for squaring. Use:

$$x \cdot y = (x+y)^2 - (x-y)^2 \ \ /2\ /2$$

- In engineering a problem is “reduced” when broken up.

  In computing “reduction” means solving problem $\mathcal{P}$

  by mapping its instances to those of a problem $\mathcal{Q}$.

  The intended lesson is that $\mathcal{Q}$ is at least as informative as $\mathcal{P}$..

Example: **INTEGER-PARTITION** *and* **EXACT-SUM**

- **INTEGER-PARTITION**:

  Instances: Finite $S \subseteq \mathbb{N}$

  Property: Exists $P \subset S$ s.t. $\Sigma P = \Sigma S/2$.

- **EXACT-SUM**:

  Instances: Finite $S \subset \mathbb{N}$ and a target $t \in \mathbb{N}$

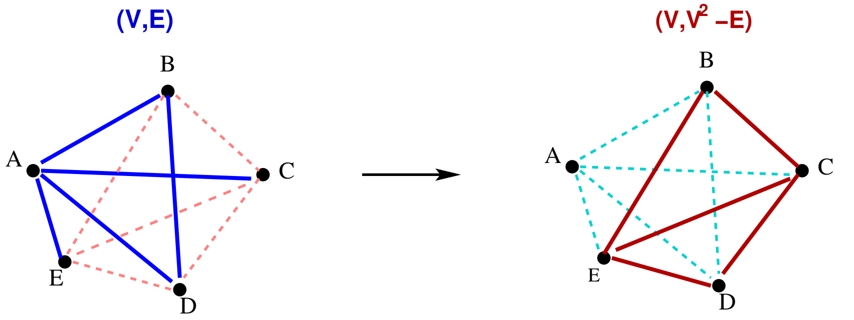
  Property: Exists $P \subset S$ s.t. $\Sigma P = t$

- Reduction ρ:

  Map instance S if **INTEGER-PARTITION** to $(S,\ (\Sigma S)/2)$

Example: CLIQUE *reduces to* INDEPENDENT SET

- A ***clique*** in graph $\mathcal{G} = (V, E)$ is a set of vertices all adjacent to each other.
- **CLIQUE**: Given $t \in \mathbb{N}$ is there a a clique of size t in $\mathcal{G}$.
- An ***independent set*** in $\mathcal{G} = (V, E)$ is a set of vertices all non-adjacent to each other.
- **INDEP-SET**: Given $t \in \mathbb{N}$ is there an independent-set of size t in $\mathcal{G}$.
- **CLIQUE** reduces to **ind-set** by a “reverse-video” mapping:



A blue graph
Missing edges are in pink
{A,B,D} a clique of size 3

A red graph
Missing edges are in blue
{A,B,D} an ind set of size 3

Reductions between problems

- A **reduction** of a decision-problem $\mathcal{P}$ to a problem $\mathcal{Q}$ is a function

$$\rho : \{\textit{Instances of } \mathcal{P}\} \rightarrow \{\textit{Instances of } \mathcal{Q}\}$$

such that $X \in \mathcal{P}$ iff $\rho(X) \in \mathcal{Q}$.

That is, if $X \in \mathcal{P}$ then $\rho(X) \in \mathcal{Q}$

and if $X \notin \mathcal{P}$ then $\rho(X) \notin \mathcal{Q}$.

- We write then $\rho : \mathcal{P} \leqslant \mathcal{Q}$.
- Such ρ is valuable when it easier to compute $\rho(X)$ than to decide $X \in \mathcal{P}$.
- When ρ is computable we write $\rho : \mathcal{P} \leqslant_c \mathcal{Q}$ and say that $\mathcal{P}$ **computably-reduces** to $\mathcal{Q}$.

Example: HALTING *computably-reduces to* ACCEPTANCE

- Define $\rho :$ HALTING $\leqslant_c$ ACCEPTANCE
- ρ maps (M, w) to (M', w)

  where M' on input w simulates M on w

  but accepts if and when M halts.
- This is a reduction:
  - If M halts on w then M' accepts w.
  - If M diverges on w then so does M',

    so it does not accept.
- ρ merely tinkers with transitions, so it is computable.

ACCEPTANCE *computably-reduces to* HALTING

- **ACCEPTANCE**: Given M & w, does M ***accept*** w?
- **HALTING**: Given M & w, does M ***halt*** on w ?
- Define $\rho :$ **ACCEPTANCE** $\leqslant_c$ **HALTING**

  ρ maps (M, w) to (M', w)

  where M' is like M with rejection converted to looping:
- M' on input w simulates M but

  enters a vacuous loop when M terminates without accepting.

Accept: If M accepts w then M' halts (and accepts) w.

Reject: If M halts without accepting then M' does not halt.

Diverge: If M diverges then so does M' .

- The reduction merely tinkers with transitions, so it is computable.

Example: ε-ACCEPTANCE *reduces to* TOTALITY

- Define a computable reduction $\rho :$ **ε-ACCEPT** $\leqslant_c$ **TOTALITY**.
- Map instance M of ε-ACCEPT to instance M' of TOTALITY
  so that M accepts ε iff M' accepts every string.
- Define M' to be the acceptor that runs M on ε and accepts x if and when M accepts ε .
- If M accepts ε then M' accepts every string.
  Otherwise M' accepts no string.
- I.e. M accepts ε iff $\rho(M')$ is total.
- The reduction ρ is computable, because it consists in
  a simple syntactic construction of an algorithm M' from an algorithm M and a string w .

Example: ACCEPTANCE $\leqslant_c$ TOTALITY & ε-ACCEPT

- Define a computable reduction $\rho :$ **ACCEPTANCE** $\leqslant_c$ **TOTALITY**.
- Map instance (M, w) of ACCEPTANCE to instance M' of TOTALITY
  so that M accepts w iff M' accepts every string.
- Define M' to be the acceptor that on input x
  runs M on w, and accepts x if and when M accepts w.
- If M accepts w then M' accepts every string.
  Otherwise M' accepts no string.
- I.e. M accepts w iff $\rho(M')$ is total.
  We also have that M accepts w iff $\rho(M')$ accepts ε.
- The reduction ρ is computable, because it consists in
  a simple syntactic construction of an algorithm M' from an algorithm M and a string w.

Composing reductions

- *If functions* $f, g : \Sigma^* \to \Sigma^*$ *are computable, the so is* $f \circ g$.
- **Proof.** The output of f is fed to g as input.
- **Theorem**

  If $\rho : \mathcal{P} \leqslant_c \mathcal{Q}$ *and* $\rho' : \mathcal{Q} \leqslant_c \mathcal{R}$ *then* $\rho \circ \rho' : \mathcal{P} \leqslant_c \mathcal{R}$.
- $\rho \circ \rho'$ is computable.

  It is a reduction:

$$\begin{array}{lll} x \in \mathcal{P} & \text{IFF} \quad \rho(x) \in \mathcal{Q} & \text{(since } \rho \text{ is a reduction)} \\ & \text{IFF} \quad \rho'(\rho(x)) \in \mathcal{R} & \text{(since } \rho' \text{ is a reduction)} \end{array}$$

Reductions preserve decidability

- **Theorem.** ***Suppose $\rho : \mathcal{P} \leqslant_c \mathcal{Q}$. If $\mathcal{Q}$ is decidable then so is $\mathcal{P}$.***
- Proof. To decide whether $X \in \mathcal{P}$
  compute $\rho(X)$ and run the decider for $\mathcal{Q}$ on $\rho(X)$ as input.
- Consequence: Show that a problem $\mathcal{P}$ is ***not decidable***
  by defining $\rho : \mathcal{Q} \leqslant_c \mathcal{P}$ for an undecidable $\mathcal{Q}$.

UNDECIDABILITY

A non-recognized problem

- The problem “Self non-accept” (**SNA**) is:

  Instances: Turing-acceptors M

  Property: M does not accept $M^{\#}$.

- We show that **SNA** is not recognized, let alone decidable.

- Suppose we had an acceptor D recognizing **SNA**, that is:

$$D \quad \text{accepts} \quad M^{\#} \quad \text{IFF} \quad M \ \text{does not accept} \ M^{\#}$$

- Taking for M the particular acceptor D :

$$D \quad \text{accepts} \quad D^{\#} \quad \text{IFF} \quad D \ \text{does not accept} \ D^{\#}$$

- **Contradiction!** So no acceptor D for **SNA** can exist!

Analogy with Russell's Paradox

- Recall Russell's Paradox:

Define $\mathcal{R} =_{\mathrm{df}} \{x \mid x$ a set, $x \notin x\}$

That is: for any set z $z \in \mathcal{R}$ IFF $z \notin z$.

- In particular taking $\mathcal{R}$ for z: $\mathcal{R} \in \mathcal{R}$ IFF $\mathcal{R} \notin \mathcal{R}$
- $\mathcal{R}$ is a collection of sets, which cannot be admitted as a "set."

Root of the problem:

Objects x are both objects and sets.

- **SNA** is a set of acceptors, which cannot be recognized by an acceptor.

Root of the problem:

An acceptor M is both a string $M^{\#}$ and a language $\mathcal{L}(M)$.

ACCEPTANCE *is undecidable*

- **SNA** is a contrived decision problem,

  designed to bootstrap our exploration of undecidability.

- **ACCEPTANCE** is a natural and important problem:

  Instances: Pairs (M, w) , M an acceptor, w a string.

  Property: M accepts w.

- **THEOREM:** **ACCEPTANCE** is undecidable.

- **PROOF:** We have **SNA** $\leqslant_c$ **NON-ACCEPTANCE** :

  Map instance M of **SNA**

  to instance $(M, M^{\#})$ of **NON-ACCEPTANCE**.

- If **ACCEPTANCE** were decidable,

  then so would be its complement **NON-ACCEPTANCE**,

  and therefor also **NSA**.

SEMI-DECIDABILITY

Semi-decidable problems

- **ACCEPTANCE** is undecidable,

  but it is ***recognized*** by an acceptor: the ***universal interpreter!***

- That's more than we can say about **NSA**!

- A problem is ***semi-decidable (SD)*** if it is recognized (as a language) by a Turing-acceptor.

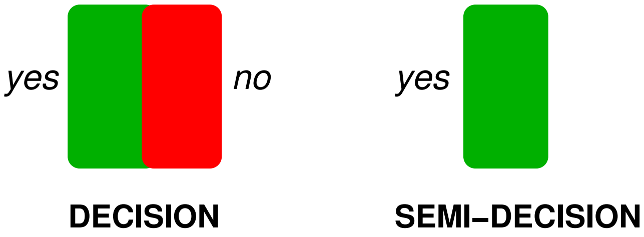
- A ***decision*** algorithm for problem $\mathcal{P}$

  identifies correctly both ***yes*** and ***no*** instances.

  A ***recognition*** (semi-decision) algorithm for $\mathcal{P}$

  identifies correctly the ***yes*** instances,

  but might loop for the ***no*** instances.



Typical semi-decision: Unbounded exhaustive search

- **INTEGER POLYNOMIALS**

  Given a polynomial $P(\vec{x})$ *with integer coefficients,*

  does it have an integer zero: $P(\vec{n}) = 0$.

- Semi-decision algorithm:

  Exhaustive Search: Try successively all tuples $\vec{n}$,

  accept P if and when such a solution is found.

Certificates for semi-decidability

- Many decision problems are of the form

  Given an instance X is there an object c such that ... ?

- Examples:

  (a) Given a graph X, is there a cycle c visiting each vertex once?

  (b) Given a natural number X, does it have a divisor $c > 1$.

- We say that c is a ***certificate*** for $X \in \mathcal{P}$.

  The cycle is a certificate for (a), a divisor is a certificate for (b).

- If the object c is provided by some benevolent power,

  it only remains to check that it actually works:

  the suggested list of vertices for (a) is indeed a cycle in the graph,

  the suggested divisor for (b) is in fact a divisor of the given integer.

Formal definition of certification

- Let $\mathcal{L}(P)$ be a problem.

  A **certification for P** is a mapping $\vdash_{\mathcal{P}}$

  from finite discrete objects (coded as strings) to instances of $\mathcal{P}$

  such that

$$X \in \mathcal{P} \quad \text{IFF} \quad c \vdash_{\mathcal{P}} X \text{ for some } c$$

Formal definition of certification

- Let $\mathcal{L}(P)$ be a problem.

  A **certification for P** is a mapping $\vdash_{\mathcal{P}}$

  from finite discrete objects (coded as strings) to instances of $\mathcal{P}$

  such that

  $$X \in \mathcal{P} \quad \text{IFF} \quad c \vdash_{\mathcal{P}} X \ \text{ for some } c$$

- The subscript in $\vdash_{\mathcal{P}}$ is omitted

  when $\mathcal{P}$ is evident.

Examples

- **COMPOSITENESS:**

  A certificate for “n is composite” is a divisor of n.

Examples

- **COMPOSITENESS:**

  A certificate for “n is composite” is a divisor of n.

- **INTEGER POLYNOMIALS:**

  A certificate for $P[\vec{x}]$ is a vector $\vec{n}$ of integers such that $P[\vec{n}] = 0$.

Examples

- **COMPOSITENESS:**

  A certificate for “n is composite” is a divisor of n.

- **INTEGER POLYNOMIALS:**

  A certificate for $P[\vec{x}]$ is a vector $\vec{n}$ of integers such that $P[\vec{n}] = 0$.

- **INTEGER PARTITION**:

  A certificate for a finite $S \subset \mathbb{N}$
  is a set $P \subseteq S$ satisfying $\Sigma P = (\Sigma S)/2$.

Decidable certifications

- A certification $\vdash$ for a problem $\mathcal{P}$ is $\boxed{\textit{decidable}}$

  if it is decidable as a set:

  There is an algorithm deciding,

  given string c and instance X whether $c \vdash X$.

Decidable certifications

- A certification $\vdash$ for a problem $\mathcal{P}$ is ***decidable***

  if it is decidable as a set:

  There is an algorithm deciding,

  given string c and instance X whether $c \vdash X$.

- Example: **ACCEPTANCE** has the certification $c \vdash (M, w)$

  where c is an accepting trace of M for input w.

Decidable certifications

- A certification $\vdash$ for a problem $\mathcal{P}$ is $\boxed{\textit{decidable}}$

  if it is decidable as a set:

  There is an algorithm deciding,

  given string c and instance X whether $c \vdash X$.

- Example: **ACCEPTANCE** has the certification $c \vdash (M, w)$

  where c is an accepting trace of M for input w.

- This certification is decidable:

  given string c and instance $(M^{\#}, w)$ of **ACCEPTANCE**

  it is easy to check that c is an accepting trace of M

  for input w .

Decidable certification = semi-decidable

- **THEOREM.** L *is recognized by an acceptor iff quad iff it has a decidable certification.*

Decidable certification = semi-decidable

- **THEOREM.** L *is recognized by an acceptor iff*

  quad iff it has a decidable certification.

  Proof of $\Rightarrow$:

  Suppose $L = \mathcal{L}(M)$.

  Let $c \vdash w$ iff c is a trace of M that accepts w.

Decidable certification = semi-decidable

- **THEOREM.** L *is recognized by an acceptor iff*

  quad iff it has a decidable certification.

  Proof of $\Rightarrow$:

  Suppose $L = \mathcal{L}(M)$.

  Let $c \vdash w$ iff c is a trace of M that accepts w.

  - $\vdash$ is a certification for L, since M recognizes L.

Decidable certification = semi-decidable

- **THEOREM.** L *is recognized by an acceptor iff quad iff it has a decidable certification.*

Proof of $\Rightarrow$:

Suppose $L = \mathcal{L}(M)$.

Let $c \vdash w$ iff c is a trace of M that accepts w.

- ▸ $\vdash$ is a certification for L, since M recognizes L.
- ▸ $\vdash$ is decidable:

  Check c's frst cfg is M's initial cfg for input w.

  Check that successive transitions in c is correct for M.

  Check c's last cfg is accepting for M.

Decidable certification = semi-decidable

- **THEOREM.** L *is recognized by an acceptor iff*

  quad iff it has a decidable certification.

Proof of $\Leftarrow$:

Suppose $\vdash$ is a decidable certification for L.

Here is an algorithm that recognizes L:

Decidable certification = semi-decidable

- **THEOREM.** L *is recognized by an acceptor iff*

  quad iff it has a decidable certification.

Proof of $\Leftarrow$:

Suppose $\vdash$ is a decidable certification for L.

Here is an algorithm that recognizes L:

- ▸ Given $w \in L$ check successive strings c

  (under size+lexicographic order) whether $c \vdash w$.

Decidable certification = semi-decidable

- **THEOREM.** *L is recognized by an acceptor iff quad iff it has a decidable certification.*

Proof of $\Leftarrow$:

Suppose $\vdash$ is a decidable certification for L.

Here is an algorithm that recognizes L:

- Given $w \in L$ check successive strings c (under size+lexicographic order) whether $c \vdash w$.
- Accept w if and when such a c is found.

Computably enumerated problems

- A problem $L \subseteq \Sigma^*$ is **computably enumerated (CE)** if there is a computable function $f : \mathbb{N} \to \Sigma^*$ with image L.
- A language $L \subseteq \Sigma^*$ is **orderly enumerated** if there is a computable ***injection*** $f : \mathbb{N} \to \Sigma^*$,

  where $|f(n)| \leqslant |f(n+1)|$,

  whose image is L.
- That is, $L = \{f(0), f(1), \ldots\}$ is a listing of L without repetition, in non-size-decreasing order.

Enumeration of decidable languages

- **THEOREM.**

  An infinite language is decidable iff it is orderly-enumerated

- $\Rightarrow$: Suppose L is recognized by a decider M.
  - Referring to size-lexicographic ordering:

    L is orderly-enumerated by

    $$\begin{aligned} f(0) &= \text{first } w \text{ accepted by } M \\ f(n+1) &= \text{first } w \text{ after } f(n) \text{ accepted by } M \end{aligned}$$
  - f is a non-size-decreasing injection by dfn,

    and is computable since M is a decider.
  - Since L is infinite, f is total.

- $\Leftarrow$:

  Suppose L is orderly-enumerated by $f : \mathbb{N} \rightarrow \Sigma^*$.

  1. Then $L = \mathcal{L}(M)$, where M is the following acceptor:

     on input w compute $f(n)$ for successive n's,

     accept if w is reached, stop and reject if $|w|$ is exceeded.

  2. M is a decider because f is

     total, injective, and non-size-decreasing.

Computable enumeration = recognition

Theorem. *A non-empty problem is recognized (i.e. SD) iff it is finite or computably enumerated.*

- $\Rightarrow$:

  Suppose L is an infinite recognized problem.

  - Then it has a decidable certification $\vdash$.
  - Since $\vdash$ is an infinite decidable set it is order-enumerated:

    $$(c_1, w_1),\ (c_2, w_2),\ \ldots$$

  - So $w_1, w_2 \ldots$ is a computable enumeration of L.

- $\Leftarrow$:

  Suppose L is enumerated by a computable $f : \mathbb{N} \to \Sigma^*$.

  - $L = \mathcal{L}(M)$ where M is the acceptor that on input w calculates $f(0), f(1), f(2) \ldots$, and accepts w if and when it is obtained as output.

Decidability in terms of semi-decidability

- We characterized SD in terms of decidability:

  L *is SD iff it has a decidable certification.*

- We now characterize decidability in terms of semi-decidability.

- **Motivation:** A decision algorithm answers **yes/no** correctly.

  A semi-decision algorithm uields just the **yes** answers.

- Decidability of L is like having two semi-decision algorithms:

  one for L and the other for $\bar{L}$.

Theorem. *A language $L \subseteq \Sigma^*$ is decidable iff both L and its complement $\bar{L} = \Sigma^* - L$ are SD.*

- $\Rightarrow$: If L is decidable, then so is its complement.

  Every decidable language is trivially SD, so both L and $\bar{L}$ are SD.

  - L is SD, so it is the image of a computable $f^+ : \mathbb{N} \to \Sigma^*$.
  - $\bar{L}$ is also SD, so it too is the image of a computable $f^- : \mathbb{N} \to \Sigma^*$.
  - To decide $w \in L$ calculate $f^+(0), f^-(0), f^+(1), f^-(1)...$ until w is obtained as an output.

  If it is an output of f^+ then $w \in L$, if of f^- then $w \in \bar{L}$.

- A problem whose complement is SD is said to be ***co-SD.***

  So the Theorem states that

  a problem is decidable iff it is both SD and co-SD.

Summary of characterizations

Let $L \subseteq \Sigma^*$

- The following are equivalent:
  - (a) L is ***semi-decidable,*** i.e. recognized by an acceptor
  - (b) L is computably-enumerated
  - (c) L has a decidable certification
- The following are equivalent:
  - (a) L is ***decidable,*** i.e. recognized by a terminating acceptor
  - (b) L is orderly-enumerated
  - (c) L is both SD and co-SD
- (a) are characterizations in terms of machine acceptors,

  (b) in terms of generators,

  (c) decidability and decidability in terms of each other.

Set operations on SD problems

- Let $L_0, L_1 \subseteq \Sigma^*$ be SD.
- **Claim:** $L_0 \cup L_1$ is SD.
- We can't just run the two acceptors sequentially:

  the first may fail to terminate.
- But since $L_0,\ L_1$ are SD, they have decidable certifications,

  say $\vdash_0$ for L_0 and $\vdash_1$ for L_1.

  Let $c \vdash w$ just in case $c \vdash_0 w$ or $c \vdash_1 w$ (i.e. $\vdash$ is $(\vdash_0 \cup \vdash_1)$)
- $\vdash$ is decidable, as the union of decidable sets.
- $\vdash$ is a certification for $L_0 \cup L_1$:

$$
\begin{array}{lll}
x \in L_0 \cup L_1 & \text{IFF} & x \in L_0 \text{ or } x \in L_1 \\
 & \text{IFF} & \text{for some } c : c \vdash_0 x \text{ or } c \vdash_1 x \\
 & \text{IFF} & c \vdash x \text{ for some } c \qquad\qquad \text{(dfn of } \vdash\text{)}
\end{array}
$$

SD is not closed under complement!

- We have seen: **acceptance** is SD but not decidable.
- If the complement of **acceptance** were SD,

  then **acceptance** would be both SD and co-SD,

  and therefore decidable, which it is not.

Proving SD via computable reductions

- We know that the problem **accept**, referring to Turing acceptors, is SD.
- There is an algorithm for transforming Turing acceptors M to equivalent general grammars G, that is such that $(\mathcal{G}) = \mathcal{L}(\mathcal{M})$.

So the following problem is also SD.

generate: *Given a grammar G and a string w, does G generate w.*

SCOPE PROPERTIES OF COMPUTING DEVICES

Decidable problems of Turing machines

- Properties of Turing acceptors may be decidable:

  Runs more than 4 steps on input `001`

  Has more than 4 states

  The accept state is the only terminal state

- These refer to the inner workings of the Turing machine

  not to the language it recognizes.

- The **ε-ACCEPT** problem is different:

  It is about the language L recognized, not the recognizing device.

- The answer yes/no would be the same for any acceptor for L.

Scope-properties of machines

- Many important properties of computing devices M are $\boxed{\textit{scope-properties}}$, in that they are about ***what*** M does, and not about ***how*** it does it.
- So a scope-property of ***acceptors*** M is a property of the language that M recognizes, i.e. $\mathcal{L}(M)$.
- If two acceptors recognize the same language then they share every scope-property.

Scope-properties of machines

- Many important properties of computing devices M are $\boxed{\textit{scope-properties}}$, in that they are about ***what*** M does, and not about ***how*** it does it.
- So a scope-property of ***acceptors*** M is a property of the language that M recognizes, i.e. $\mathcal{L}(M)$.
- If two acceptors recognize the same language then they share every scope-property.
- Similarly, a scope-property of ***transducers*** M is a property of the partial-function that it computes.
- If two transducers compute the same partial-function then they share every scope-property.

Scope-properties of machines

- Many important properties of computing devices M are **scope-properties**, in that they are about ***what*** M does, and not about ***how*** it does it.
- So a scope-property of ***acceptors*** M is a property of the language that M recognizes, i.e. $\mathcal{L}(M)$.
- If two acceptors recognize the same language then they share every scope-property.
- Similarly, a scope-property of ***transducers*** M is a property of the partial-function that it computes.
- If two transducers compute the same partial-function then they share every scope-property.
- Scope-properties are also called by logicians *scope, extensional, index-sets* and *semantical*.

Examples for Turing-acceptors

- $\mathcal{L}(M)$ is finite.
- $\mathcal{L}(M)$ is infinite.
- Accepts at least two strings, i.e. $\mathcal{L}(M)$ $\geqslant 2$ elements.
- Every string accepted by M has even length.
- $\mathcal{L}(M)$ is a regular language.

  This does ***not*** mean that M is a DFA.
- For some $n > 0$ M accepts every string of length n.
- For every $n \geqslant 0$ M accepts some string of length n.

Examples for Turing-transducers M .

- Computes a total function.
- Undefined for input ε.
- Define for all input of even length.
- Undefined for all input of even length.
- Constant (same output for all input
- Increasing: If $|x| < |y|$ then $|f(x)| < |f(y)|$
- Bounded: There is an $n \in \mathbb{N}$ s.t. $|f(x)| \leqslant n$ for all x .
- Unbounded: For every n there is some x s.t. $|f(x)| > n$.
- Inflationary: $|f(x)| \geqslant |x|$ for all x.

Non-scope properties of Turing machines

- Has more than 100 states.
- Reads every input to its end.
- For some input visits every state during computation
- Never runs more than n^2 steps for input of size $\leqslant n$
- Is a decider

  (but “recognizes a deciable language” ***is*** a scope-property!)

Rice's Theorem

- A property is ***trivial*** for a language L

  if it is either true of every $w \in L$ or false for every w .

- Example: The property $\mathcal{L}(M)$ ***is SD***

  is always true: it just conveys the definition of SD.

- **Theorem.** (Henry Rice, 1951).

  There is no decidable scope-property of Turing-acceptors, other than the trivial properties.

- **Proof idea:**

  If $\mathcal{P}$ is non-trivial, then **ε-ACCEPT** $\leqslant_c \mathcal{P}$.

  So $\mathcal{P}$ is undecidable.

Proof of Rice's Theorem

- Let $\mathcal{P}$ be a non-trivial scope-property of Turing acceptors.

  Fix some acceptor E recognizing $\emptyset$.

  Assume $E \notin \mathcal{P}$ (it won't matter).

  Also, $\mathcal{P}$ is non-trivial, so it is true of some acceptor A.

- Note: we have E and A on opposite sides of $\mathcal{P}$!

- Define $\rho : \boldsymbol{\varepsilon} - \textbf{ACCEPT} \leqslant_c \mathcal{P}$,

- Write M' for $\rho(M)$.

- On input x, M' disregards x, and runs M on ε.

  If and when M accepts $\boldsymbol{\varepsilon}$, M' fires A on x.

- So we have

$$\mathcal{L}(M') = \text{ if } M \text{ accepts } \boldsymbol{\varepsilon} \text{ then } \mathcal{L}(A)$$
$$\text{else } \emptyset\text{, i.e. } \mathcal{L}(E)$$

- SO M accepts ε just in case $M' = \rho(M) \in \mathcal{P}$.
- The reduction consists in tinkering with transitions, so it is computable.

Using Rice's Theorem

- All previous examples of scope properties are non-trivial, and therefore undecidable.
- Rice's Theorem says nothing about problems that are are not scope-properties.
- Properties referring to the structure of Turing machines, or to the syntax of programs, are always decidable.
- But properties that relate to the workings of algorithms and machines are often, but not always, undecidable.
- Example. **ALL-STATES-USED**:

  Each state of a given TM M occurs in some trace of M .
- It is not hard to show that ε-**ACCEPT** $\leqslant_c$ **ALL-STATES-USED**.

  So the non-scope problem **ALL-STATES-USED** is undecidable.

- Outline of the reduction:
  - Acceptor M is mapped to M', which uses an additional alphabet symbol $\#$ and an additional state t.
  - M' runs M on input ε and if accepted writes $\#$, switches to t, and cycles while reading $\#$ through all states of M.
- So if M accepts ε then M' uses all its states. If not, then M' does not use the state t.

SUMMARY OF METHODS

Proving problems to be decidable

Methods for proving problems L ***decidable:***

- L is recognized by a decider.
- L is finite or orderly-enumerated.
- Both L and $\overline{L}$ are semi-decidable.
- L is definable using union, intersection,
  and complement (or difference) from decidable problems.
- L is computably reducible to a decidable problem.

Proving problems to be SD

Methods for proving problems L ***semi-decidable:***

- L is recognized by an acceptor.
- L is computably-enumerated.
- L has a decidable certification.
- L is defined using union and intersection from SD languages.
- L is computably reducible to a SD problem.

Proving properties to be undecidable

- Methods for proving problems L ***undecidable:***
    - L is a stable property of acceptors or transducers.
    - L is defined using complement from undecidable languages.
    - An undecidable problem is computably reducible to L.
- Methods for proving problems L ***non-SD:***
    - L is the complement of a SD but undecidable problem.
    - A non-SD problem, such as ε**-NON-ACCEPT**, is $\leqslant_c L$.

OTHER UNDECIDABLE PROBLEMS

String equations

- Recall string expressions:

  generated from variables and fixed strings

  using the concatenation, head, and tail operations.

- Solution to $E = E'$:

  a binding of variables to strings, for which the equation is true.

- $x * \mathtt{01} * y = y * \mathtt{10} * x$ has as solution $x = \mathtt{11},\ y = \mathtt{1}$.

- **string-equation** Problem:

  Given an equation between string-expressions, does it have a solution?

- **string-equation** is ***undecidable***

Integer equations

- **Arithmetic expressions:** generated from variables and numerals using $+$ and $\times$.
- A solution to $E = E'$:

  a binding of variables to integers, for which the equation is true.
- **arith-equation** Problem:

  Given an equation between arithmetic-expressions, does it have a solution?
- Proved undecidable in the 1970's, incrementally,

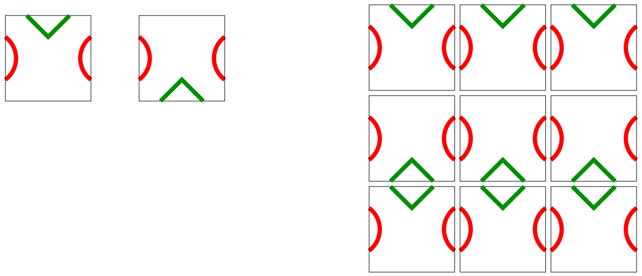
  by Yuri Matiyasevich, Julia Robinson, Hilary Putnam and Martin Davis.

⋆*Tessellation*

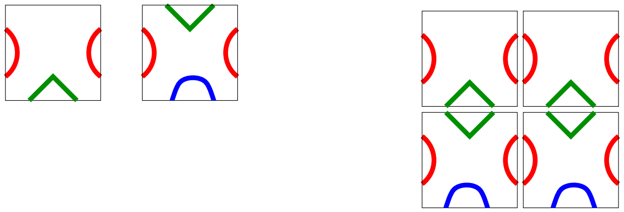
- Consider square tiles, with each side marked with a design.

  Such tiles are used to tessellate rectangles, with similar abutting sides.

- Example: The following tiles can be used to tessellate a 3×3 display



- The following tessellates a 2×2 display, but not any 3×3:



- Undecidable: The **tiling** Problem (Hao Wang, 1961):

  Given a set P of marked tiles,

  can P tessellate arbitrarily large rectangles.

**Post's Correspondence Problem*

- A ***correspondence*** over an alphabet Σ is a finite set of pairs

$$(u_1, v_1), \ldots, (u_k, v_k) \qquad (u_i, v_i \in \Sigma^*)$$

- A ***match*** for such a correspondence is a string w
  that can be read both as a concatenation of some u_i's
  and as the concatenation of the *corresponding* v_i's:

$$w = u_{i_1} \cdots u_{i_n} = v_{i_1} \cdots v_{i_n}$$

- Example: The correspondence $C = \{(100, 1), (0, 100), (1, 00)\}$
  has the following match:

$$\begin{aligned} w &= 100\ 1\ 100\ 100\ 1\ 0\ 0 \\ &= 1\ 00\ 1\ 1\ 00\ 100\ 100 \end{aligned}$$

- The undecidable **post's correspondence** Problem:
  Given a correspondence C, does it have a match?

⋆*The Perishable Matrix Problem*

- A finite set S of $k \times k$ matrices is ***perishable*** if some product of matrices out of S (repetition allowed) yields the zero $k \times k$ matrix.
- **PERISHABLE-MATRIX** Problem:
  Given s set of $k \times k$ integer matrices (for some k), is it perishable.

Problems about CFGs

- Does a given CFG over Σ generate all Σ-strings? (Whether a CFG generate some string ***is*** decidable).
- Is a given CFG ambiguous?
- Given CFGs G and G', is $\mathcal{L}(G_0) \subseteq \mathcal{L}(G_1)$?
  Is $\mathcal{L}(G) \cap \mathcal{L}(G')$ empty?

⋆ *Validity*

- Relational Logic gives rules of reasoning for the basic logical operations: the connective: $\neg, \wedge, \vee$, and the quantifiers: $\forall, \exists$
- Example: $(\forall x\ P(x))\ \vee\ (\exists y\ \neg P(y))$.
- A statement is ***valid*** if it is true regardless of the particulars.

  For example, the statement above is valid.
- In contrast, the following statement is not valid:

$$(\forall x\, \exists y\ P(x,y)) \vee (\exists x\, \forall y \neg P(x,y))$$

- The **VALIDITY** Problem:

  Given a statement of relational logic, is it valid?
- Undecidability proved in 1936

  independently by Alan Turing and Alonzo Church.